MODULE INTERCONNECTION LANGUAGES

Need a better introductory paragraph.

RUBEN PRIETO-DIAZ
JAMES M. NEIGHBORS

December 1983

Department of Information and Computer Science

University of California Irvine

Irvine, CA 92717

This work was supported by the U.S. National Science Foundation under Grant MCS-81-03718 and by the Consejo Nacional de Ciencia y Tecnología (CONACyT), MEXICO.

Table of Contents

- Ada?

Don't need to include those having similar properties

List of Figures

1. INTRODUCTION

DeRemer and Kron developed the first Module Interconnection Language [DeRemer & Kron 76]; MIL75. They established the basic ideas and concepts of module interconnection by arguing convincingly about the differences between programming-in-the-small for which typical programming languages are used to write modules and programming-in-the-large for which a module interconnection language is required for "knitting" those modules together.

Three MILs [Thomas 76], [Cooprider 79], and [Tichy 80] were developed after MIL75 each one adding new ideas and features to MIL75 but, essentially based on DeRemer and Kron's original concepts. Thomas developed a module interconnection notation and discussed a possible module interconnection processor, Cooprider expanded the basic ideas of MIL75 to introduce a version control facility and a software control facility, and Tichy developed INTERCOL, a MIL that integrates some of Cooprider's features with control of system families and with asynchronous compilation of modules.

There exist several languages, software development tools, and operating systems that in one way or another provide module interconnection mechanisms. To describe every tool, system or methodology that supports some kind of module interconnection is beyond the scope of this paper. Hence, the scope of this paper is to survey the languages that are specifically designed to support module interconnection and that are called Module Interconnection Languages (MILs) and to include brief descriptions of some modularization techniques and tools that support module interconnection to provide a frame of reference and a basis for comparison.

The goal of this survey is to acquaint the reader familiar with the problems

of programming and maintaining large software systems with a class of tools designed to describe how a large system "fits together". In particular we will focus on MILs designed only to specify the structure of a system rather than its behavior as is important in system modelling. We will largely not deal with how the MILs perform their functions in order to treat the issue of what they do in more detail. The interested reader will find more detail in the referenced literature.

Work in this area can be traced back to the early 1960s when the first large software systems like OS/360 started to create very difficult problems not only to their system programmers but to their system designers and project managers as well. The basic design principle used then was that of "divide and conquer". Divide the system into modules by the process of system design. Then program the modules, validate each module and, assemble all modules to integrate a complete system. This basic design principle is still the primary design technique used today.

The crux of the problem is the module assembly (integration) part of a complete system. It requires manual inspection of module interfaces in order to guarantee perfect module bonding. MILs alleviate this problem by providing certain language constructs different from regular programming languages, that succeed in representing the various module interconnection specifications required to assemble a complete software system.

A MIL can be considered a structural design language because it states what the system modules are and how they fit together to implement the system's function. This is architectural design information. MILs are <u>not</u> concerned with what the system does (specification information), how the major parts of the

system are embedded into the organization (analysis information), or how the individual modules implement their function (detailed design information).

While the major payoff of using a MIL may seem to be during the system design phase of the software-lifecycle the actual payoff is during system integration, evolution and maintenance. This is because the MIL specification of a system constitutes a written down description of the system design which must be adhered to before a version of the system may be constructed. A maintenance programmer cannot violate the system design without explicitly modifying the system design.

1.1 Current Research

Current research in module interconnection can be observed from three different but complementary perspectives: The Software Engineering Perspective, the Formal Models Perspective and, the Artificial Intelligence Perspective. The basic question in module interconnection is: given a collection of agents (modules) each of which performs a certain function under certain circumstances, how can these agents be combined to perform a more complex function?

Researchers in Software Engineering view the problem as a design problem and approach the problem from the point of view of finding a design notation which can capture the complete design of a system as stated explicitly by a system designer. MILs are design notations resulting from this point of view. The system designer is thought of as "coding" in design notations. A MIL description of a system is mechanically checked for consistency and completeness before the system is actually linked together.

Researchers working in Formal Models view interconnection in two ways: as a structural model of the resource usage of the system during execution and as a

consistency model of the construction of the system. The resource model is intended to determine the data loading of different parts of the system and to detect any communications deadlocks which might occur. The SARA system [Campos & Estrin 78a] [Campos & Estrin 78b] has adopted this structural modelling as one of its main goals. A system consistency model captures the constraints on using different versions or implementations of individual modules composed of other modules. Given these formal constraints and the modules which must be implemented, a consistency model determines a collection of specific versions and implementations of modules which can be shown to implement the system [Neighbors 80].

For the Artificial Intelligence researcher the interconnection problem manifests itself as a problem in automatic programming. In this context "knowledge about programming" or "knowledge about the problem domain" can represent both constraint and implementation information. The problem becomes one of using this knowledge base to arrive at a sequence of low-level steps which implement a high-level specification. This search for an acceptable series of steps is guided by a description of the problem to be solved (goal), hints about a series of steps which might suffice for a given goal (plan), and which plans are potentially useful in different circumstances (frame). The goals, plans and frames are all a part of the knowledge base. These mechanisms must make sure that the steps that they link together are compatible and this is the interconnection problem. The Transformational Implementation system [Balzer, Goldman & Wile 76] and the Programmer's Apprentice system [Rich, Schrobe & Waters 79] are two systems which take this approach.

The point of view taken in this survey is that of the Software Engineering

perspective. The point of view of the other two perspectives is very important and deserves a complete in depth study for each. Since each of these views is dealing with similar interconnection information it is important that a researcher taking one perspective understand the other perspectives by the information they manipulate and the operations they provide.

2. MIL CONCEPTS AND IDEAS

Modularity is a well established concept that has been used in engineering and managerial disciplines for many years to break the work of a big project up into controllable units. In both of these approaches, the details of the division have not been very important. In the design of a software system however, the splitting up is crucial. It must be done so as to minimize, to order, and to make explicit the connection between the modules. Moreover, if the aim is a testable and validated system, system connectivity must be substantially reduced.

There are no rules on how to do this, but some helpful methodological guidelines have been developed. The keynote behind these guidelines is that of hierarchical ordering as a technique to control complexity [Newell et.al. 61].

Other technique having hierarchical ordering as its aim is the idea of "successive abstraction" [Dijkstra 65], where the idea of the divide and conquer approach to characterize top-down design and the principle of non-interference were introduced.

The main idea in these early works is that of separating the behavior of the program at one level from the details of each of the components thus reducing the complexity of the programming problem. Each of the subprograms can then be considered in turn, in isolation from each other and from the program skeleton in which they are embedded.

Some of these ideas go as far back as the concept of mathematical function or even to earlier times and an exhaustive historical search of these ideas is beyond the scope of this paper.

Even though the key word "modularization" or "module" did not become widely used until the early seventies, the original work on structured programming and hierarchical system decomposition provided the conceptual background for the development of Module Interconnection Languages (MILs) of the late seventies.

MILs are based on the effective separation between Programming-in-the-large (PL) and Programming-in-the-small (PS). PS is concerned with building programs and has been greatly developed to include the new techniques of structured programming, top-down design, stepwise refinement, and others. Many of the widely accepted languages (ALGOL, PASCAL, COBOL, etc.) have been designed to aid programming-in-the-small and have contributed towards making programming a science [Gries 81]. The system lifecycle phases of detailed design and implementation primarily use PS notations. These notations focus on how a particular part (module) of a system performs its function.

PL on the other hand, is concerned with building systems. PL notations are primarily used in the architectural design phase of system construction and concentrate on how the system modules cooperate (through calls and data sharing) and what functions each module provides. A language concerned with the data and control flow interconnections between a collection of modules we will refer to as a Language for Programming in the Large (LPL). A MIL is an LPL with a formal machine-processable syntax (i.e. not natural language or graphical diagram) which provides a means for the designer of a large system to represent the overall system structure in a concise, precise, and verifiable form.

MILs are very effective but limited tools to aid during the software life-cycle. A system must be evaluated, analyzed, and designed first by means of current methods and techniques. Once a system structure is determined, it may

be coded in a MIL to be checked and verified for completeness and inconsistencies. A separate MIL code must be maintained during implementation and then used for high level maintenance during system operation and enhancement.

The main concepts of MILs could be listed as:

1. The idea of a separate language to describe system design.
2. To be able to perform static type-checking at an intermodule level of description.
3. To consolidate design and construction process (module assembly) in a single description.
4. Capability to control different versions and families of a system.

A MIL usually serves as a Project Management Tool by encouraging structuring before starting to program the details and as a Support tool for the design process by capturing overall program structure and being capable of verifying system integrity before design implementation begins. A MIL could also provide some means of standardizing communication among members of a programming team and of standardizing documentation of system structure. The significant support to these activities as seen from the Software Engineering perspective, is what makes MILs an important tool for the software development process.

Example

In a MIL description, resources are considered objects that become the currency of exchange among modules. Resources are any entity that can be named in a programming language (e.g. variables, constants, procedures, type definitions, etc.) and which can actually be made available for reference by another module within a given software system.

All resources are ultimately provided by modules, thus modules are units that

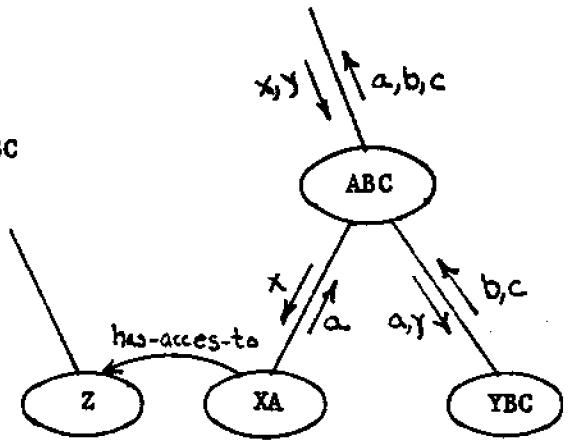
provide resources and that require some set of resources. The syntax primitives of a MIL describe the flow of resources among modules; they are provide (which may also be called synthesize or export) and require (which may also be called inherit or import). Has-access-to is another syntax primitive that helps to provide proper module structure within a system. A must attribute may also precede the above primitives.

An example of a MIL description of a module is shown in figure 2-1 below. Note that declarations such as module, function, and consist-of are also part of the MIL syntax.

```
module ABC
    provides a,b,c
    requires x,y
    consist-of function XA, module YBC

        function XA
            must-provide a
            requires x
            has-access-to  module Z
            real x, integer a
        end XA

        module YBC
            must-provide b,c
            requires a,y
            real y, integer a,b,c
        end YBC
end ABC
```



a: MIL source code b: Graphical representation

Figure 2-1: MIL Description of a Module

The MIL description of a module specifies the resources required and provided by the module and becomes the interface with other modules and subsystems. Module descriptions are the actual code of a MIL and are used when assembling or integrating a software system in order to verify system integrity.

In most of the module interconnection schemes we shall examine, the PL information is in the form of a MIL and the PS information is in the form of a normal programming language. The packaging of this information differs between different schemes. At one side of the spectrum a system is defined as a collection of modules each of which contains MIL and PS information and there is no central description of the system other than the list of modules which compose it. At the other end of the spectrum the modules which compose the system contain only PS information while the central description of the system contains all the MIL information for each module and the interconnections in the system. In both cases it makes sense to "compile" the MIL definition of a system to see if the interfaces between it's constituent parts match. No programming language (PS level) information is necessary to perform this compilation.

What MILs Don't Do

There are some functions that are not considered to belong to the domain of MILs. These functions were stated by DeRemer and Kron [DeRemer & Kron 76] and by Thomas [Thomas 76] in order to make a clear distinction between a MIL and other tools or languages performing similar functions related to module interconnection. With this separation of functions the above authors intended to state the "universe of discourse" of MILs establishing the basis upon which newer MILs should be built.

The functions a MIL should not attempt are:

1. Loading: A MIL should leave this function to a "subsystem loading language" or to other facilities within the software development environment.
2. Functional Specification: A MIL only shows the static structure of a software and should not specify the nature of its resources. This task should be assigned to other subsystems.
3. Type Specification: A MIL is concerned with showing and verifying the

different paths of communication among modules within a software system by means of named resources. Some of these resources may be types but the naming of these types is what a MIL looks for, not their specification. For example, the decision to declare real y in a program is a design decision that follows a type specification while real y in a MIL code acts as a type checking statement only.

4. Embedded Link-edit Instructions: These operations should be left for another subsystem within the development environment such as the operating system or a separate command language.

The current tendency in MIL development, is to keep the domain of MILs well defined so that stand-alone MILs can be developed and then integrated as part of a software development environment such as in GANDALF [Haberman, et.al. 81].

Approaches such as C/MESA of the MESA System [Lauer & Satterthwaite 79] and External Structure of ADAPT [Archibald 81] conform to the current tendency but are not as general since they are restricted to modules coded in a single programming language.

Modern programming environments provide tools that support module interconnection along with version control mechanisms and other software development aids (i.e. PWB, MESA, CDL2, GANDALF etc.) Some of the module interconnection tools integrated in these systems are implementations of MILs (i.e. MESA's C/MESA, SARA's MIC, GANDALF's INTERCOL, DREAM's DDN) while others are collections of specialized tools (i.e. PWB, PROTEL, CDL2).

3. MODULE INTERCONNECTION LANGUAGES

There are four stand-alone MILs developed to date and reported in the literature: MIL75 [DeRemer & Kron 76], Thomas' MIL [Thomas 76], Cooprider's MIL [Cooprider 79], and INTERCOL [Tichy 79].

Figure 3-1 shows the evolution of MILs along time. The first two MILs have a strong modular language origin with a tendency to use software engineering techniques. The last two MILs although sharing the same origins with their predecessors go further by integrating techniques from software engineering and tools from powerful operating systems. The graph represents the citations made by papers in the different areas at the given times and serves to show that the problem addressed by MILs are not confined to a single area.

DeRemer and Kron developed the first Module Interconnection Language [DeRemer & Kron 76]; MIL75. They established the basic ideas and concepts of module interconnection. MIL75 is a language for programming-in-the-large (LPL) that gives the systems designer a tool to design and, to a certain extent, build a complete system out of modules that do not have to be completely coded and tested, just properly specified. For each module the designer must specify the resources provided and required. The type of the resources must also be specified. Details about the internal operations of the modules are not required. MIL75 compiles all these specifications while doing consistency checking resulting in an accurate recording of the overall solution structure.

Thomas [Thomas 76], developed a module interconnection notation and discussed a possible module interconnection processor. He proposes a formal model based on the separation of compiling, binding and linking that allows for flexible bindings and also provides the notation to incorporate his MIL into a

programming system. Besides the flexible binding scheme which is his main contribution to MILs, Thomas presents through his formal model the basis for practical MIL implementations.

Cooprider [Cooprider 79], expands the basic ideas of the previous MILs to introduce a version control facility and a software construction facility. The former facility recognizes the different instantiations (versions) of an interconnection network and knows how they are hierarchically integrated while the latter facility is capable of constructing a complete software system from a functional description of the construction process. Resources and source files are combined according to construction rules, explicitly specified by the designer, to create the objects that form a software system.

His major contribution to MILs is to discard the use of a compiler and to use instead a data base processor (similar to the system described in [Bratman & Court 75]) supporting an interactive system construction environment.

INTERCOL was developed by Tichy in 1979 ([Tichy 79] and [Tichy 80]). In addition to the features of Cooprider's MIL, INTERCOL supports asynchronous compilation of modules and/or subsystems, and control of system families. INTERCOL is intended to be an integrated software development and maintenance environment that supports communication and cooperation among programmers. GANDALF has integrated INTERCOL as its tool for system version description and generation.

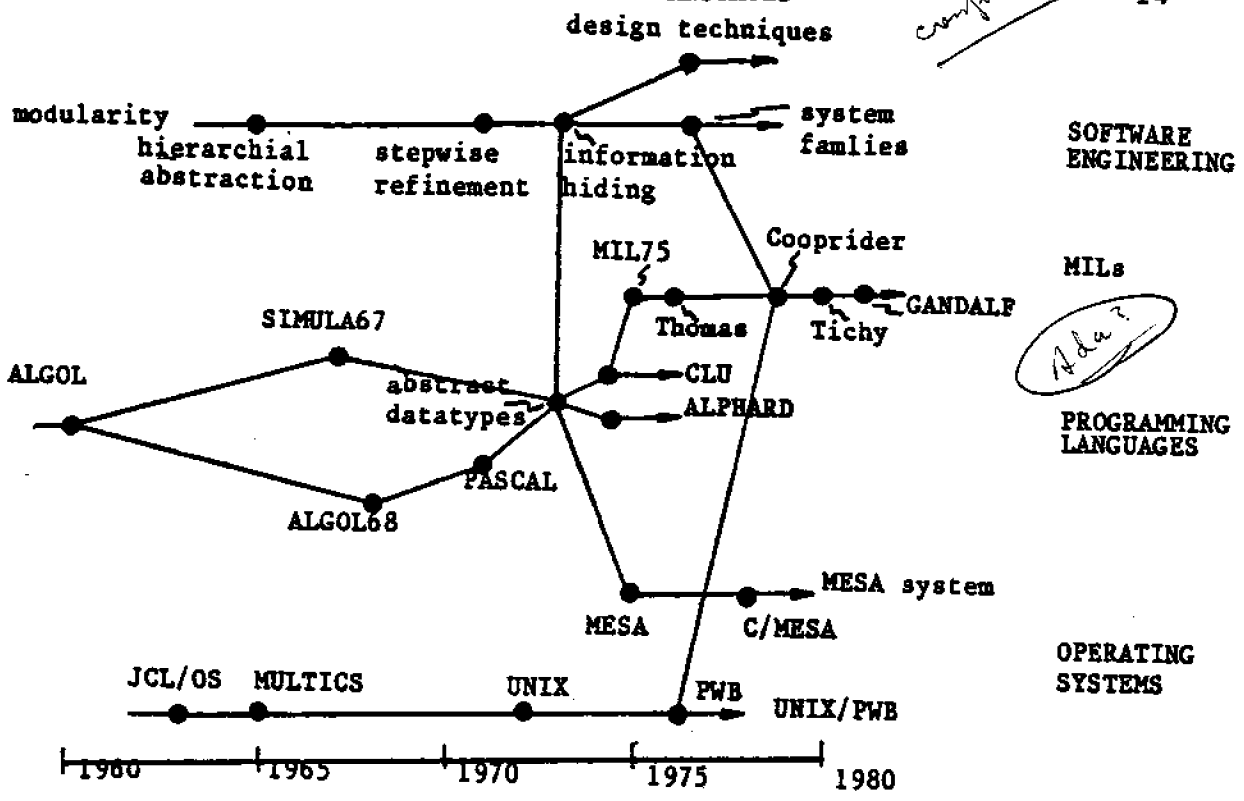


Figure 3-1: Graphic View of MIL Evolution

3.1 MIL75

MIL75 is based on the concept that any system structure has a graphical representation in the form of an inverted tree with nodes being the modules and the edges their different hierarchical relationships. This graphical relationship of a system is an implicit prerequisite to use MIL75. The methods proposed in [Stevens, et.al 74] and in [Yourdon & Constantine 79] for structured design could be used to obtain the hierarchically decomposed inverted tree representation of a system as required by MIL75, provided some additions are included to represent module accessibility as well as the resources required and provided.

Once a graphical structure for a system is obtained, it is programmed in MIL75 where the code consists of the description of the modules in each node. The code is compiled to verify system integrity and to enhance reliability.

Each "system description" can be recompiled alone or with any others. When "systems descriptions" are put together they define a "module interconnection structure".

MIL75 consists of three sets that are required to establish system structure:

1. Resources- Atomic elements which denote abstractions of programming constructs within a program (variables, types, arrays, functions, etc.) and are available for reference to other modules.

2. Modules- Programming units made up of resources and other programming constructs that perform a specified function or task.

3. Systems- Groups of hierarchically organized modules that communicate via resources to perform more elaborate functions.

MIL75 establishes certain relationships between resources and modules as the basis to keep system structure, integrity and maintainability within control. These relationships are based around the inverted tree model described above and form the minimum set required by MIL75 to be able to do module interconnection. The relationships are:

1. Defining the scope of definitions of module or subsystem names thus helping to impose the overall system structure called here the "system tree". This external scope definition is accomplished by the systems designer and the description of each node (module or subsystem) is written in MIL75 code. Thus this relationship is among modules. Figure 3-2 below shows a system tree for a one-pass compiler.

2. The relationship between modules and their provided and derived resources. This relationship is represented by a "Resource Augmented Tree" which is a system tree that also indicates the resources provided and derived for each node pursuing a top-down approach. This tree shows only the flow of resources from parent to children and up from children to parent, the latter being called "derived resources". Resources originated in other nodes not being direct ancestors or successors are not considered "provided" nor "derived" but rather "accessed" resources.

3. The relationship among the resources of sibling modules. The channels for flow of resources among siblings are determined by the parent. These accessibility channels or links among a set of siblings may form any directed graph. Access rights are not transitive and also the children of a node are invisible to its siblings. This relationship limits resource accessibility to modules

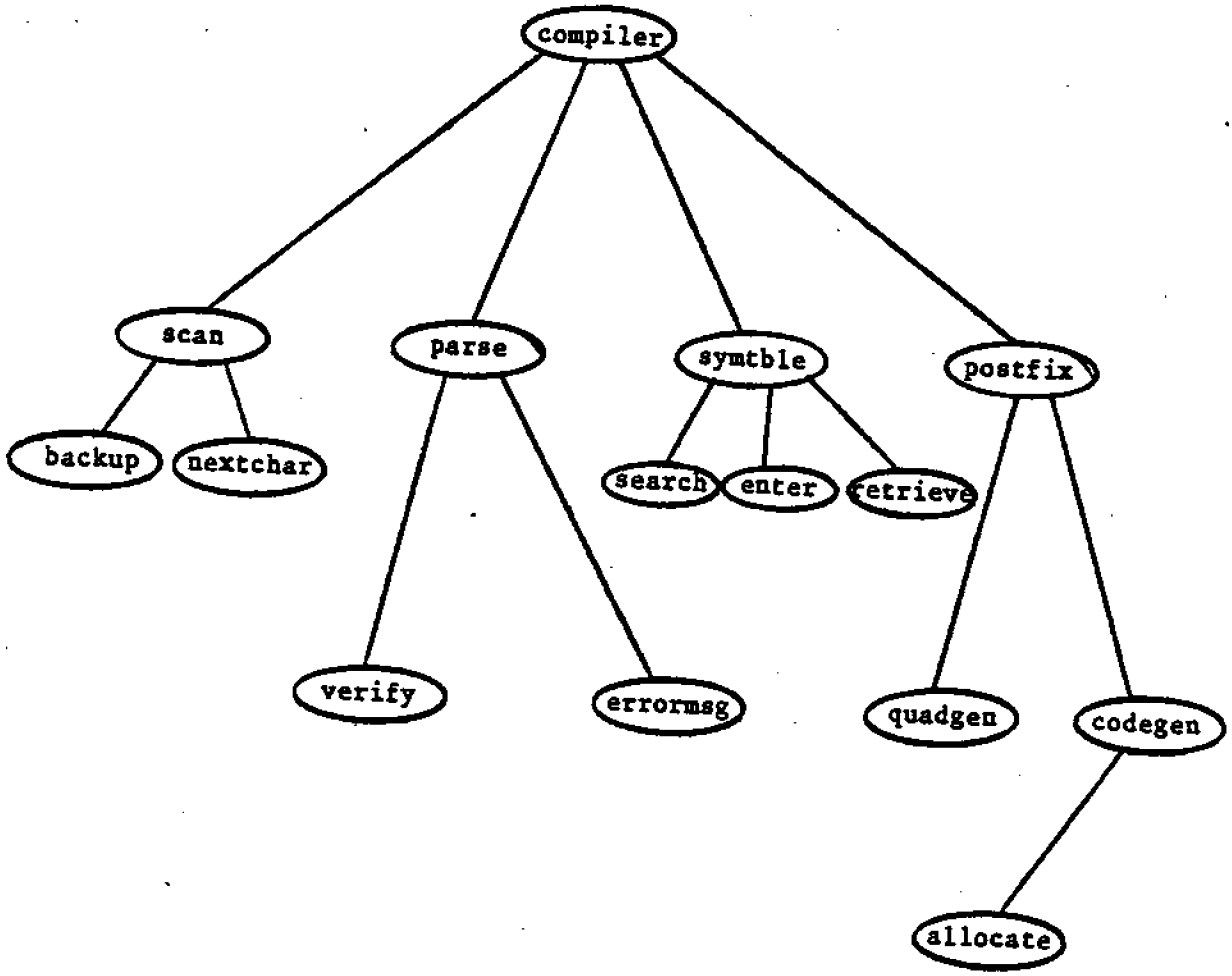


Figure 3-2: Graphical System Tree for a One-pass Compiler

laying at the same hierarchical level.

4. The relationship of accessibility of resources of modules at different hierarchical levels. On the one hand a child inherits by default all access rights that have been granted to its parent but a parent may deny some subset of its access rights to any of its children. The parent however, must explicitly list all access rights left to a "partially disinherited" child. On the other hand a parent has access to the resources that it demands from any of its children but these rights cannot be transmitted to the next level down because its grandchildren and lower descendants are invisible to the parent.

5. The relationship between modules and the origin and usage of resources. For each module, a MIL75 program must include two statements:

   a. The "statement of origin" listing the resources defined in that module and,

   b. The "statement of usage" listing separately the derived resources provided by its children, and all other resources, those obtained through siblings or inherited access.

Establishing relationships 1 through 5 is what MIL75 coding is all about. A ready-to-compile code must describe the "access augmented system tree" which is shown as part of Figure 3-3 below.

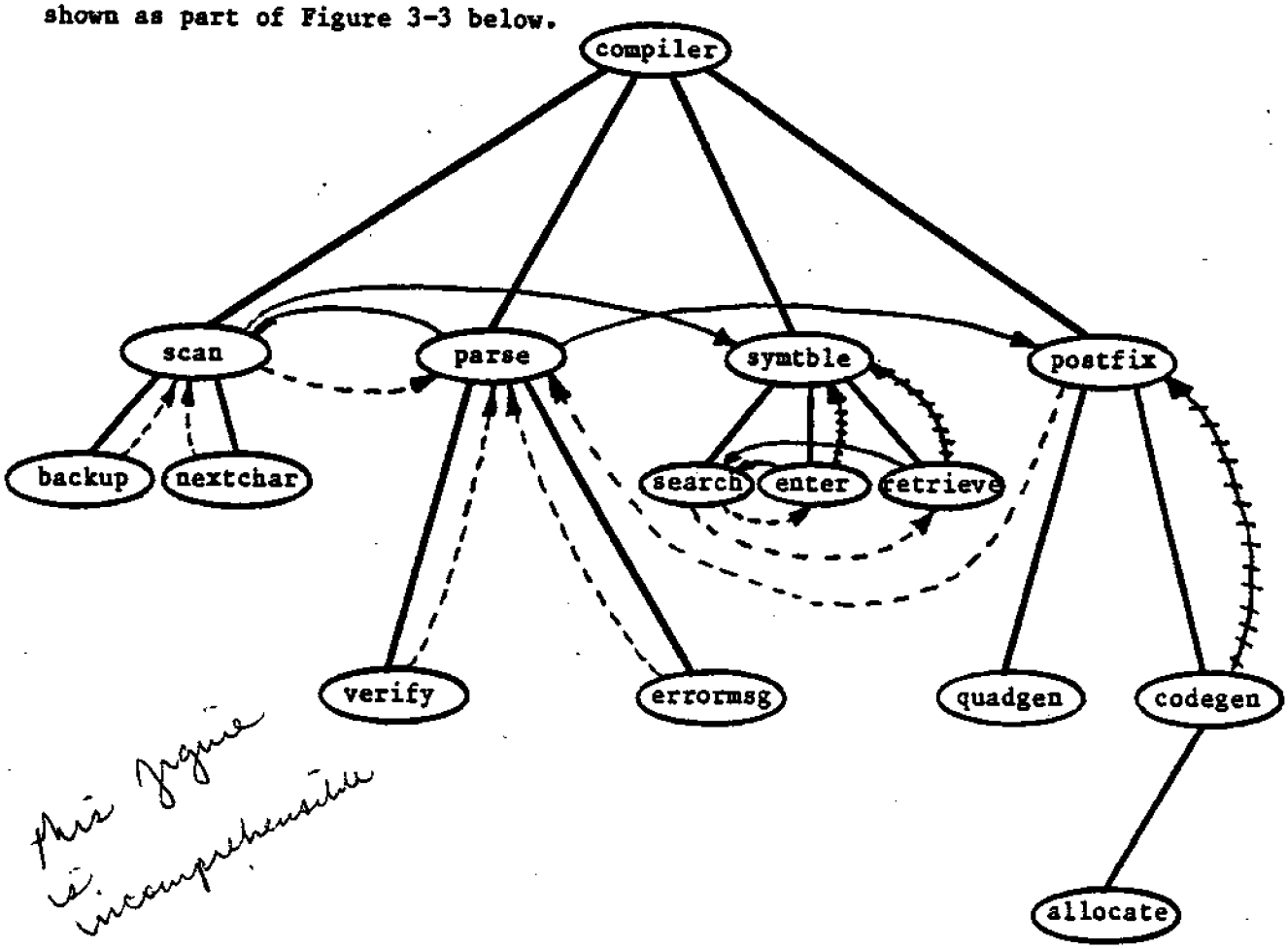


Figure 3-3: The Module Interconnection Structure

After these relationships have been established (coded) by the system designer, the MIL75 compiler checks that actual usage of resources by a given module agree to access rights provided by other modules to those resources and that provided resources either come from a child or are defined within that module. Passing that stage, the compiler then establishes the usage links which

are direct channels where resources will flow.

A usage link is illustrated as follows: if a module m has access to a resource provided by a module p then a usage link is established to point to m from p. In other words, it is solving indirect references by direct links which in short corresponds to binding (at compile time). This binding is what establishes the "module interconnection structure" shown in Figure 3-3. For this example the access augmented system tree mentioned above is identical to the module interconnection structure of Fig. 3-3 except for the usage links. In short, a complete MIL75 program consists of a series of statements expressing the different relationships (1,...,5 above) between resources and modules of a structured (nodal) representation of a system.

The partial code of the MIL75 program for the module interconnection structure shown is given in figure 3-4 below.

```
system compile
  author John Smith
  date   2/25/82
  provides compiler
  consists of
    root module
      originates compiler
    subsystem scan
      must provide scanner
      has access to symtble
      consists of
        root module
          originates scanner
          uses derived
```

Figure 3-4: Partial Code of a MIL75 Program

Differences from the Other MILs

MIL75 is oriented around a structured (oriented tree) representation of a system thus shifting some of the work back to the systems designer. The MIL compiler takes (in MIL75 code) the complete description of the system where

design decisions like proper abstraction, functional decomposition, and modularization have already been made by the systems designer. Furthermore the systems designer must establish the accessibility and provision of resources among modules.

The main contribution of MIL75 to the field of software systems design is in providing the designer with some means of detecting wrong design decisions before construction begins. If the MIL75 compiler detects an error, it may be an error reflecting a bad modularization of the system or simply an inconsistency on the flow of resources. In the later case, the fix is relatively easy and requires the recompilation of one or few modules and/or subsystems while in the former case a recompilation of the complete system may be required.

The major drawback of MIL75 is its rigidity caused by its attachment to the inverted tree structure. Thomas (next section), tried to overcome this deficiency by designing his MIL around a more flexible structure. Another deficiency in MIL75 is its lack of support for the "specification of the function of the modules". DeRemer and Kron also mention the capability a MIL should have to support modules programmed in distinct languages but they do not show such capability for MIL75. Last but not least, MIL75 could be seen as an isolated tool used only to show how a MIL should work but was not integrated into a software development environment. Thomas instead, tried to establish the mechanisms to integrate his MIL into a programming system. This integration is required, as Cooprider [Cooprider 79] and Tichy [Tichy 80] later did, in order to use and evaluate their MILS.

Experience to Date

MIL75 was implemented in an academic environment to test the concepts of module

interconnection but was never used in a production environment nor integrated into a software development system.

3.2 THOMAS' MIL

The objective in Thomas' thesis is to propose a MIL that would be a complement to CLU/ALPHARD-like languages and that would incorporate new ideas for the future development of MILs.

Thomas' MIL is based on the idea that module interconnection should be flexible and not constrained to a particular system structure as in MIL75. He advocates the "compiling and static type-checking before binding/linking" scheme and claims to obtain more flexibility, less recompilation, and moderate cost because of the composition of binding and linking into a single phase.

This scheme allows a software system to be represented (in MIL code) as a "finite directed graph G with no simple cycles and where S is a start node in G and all nodes in G are reachable from S". This graph definition is the same as the inverted tree representation of a software system used by MIL75 with the addition of cycles. Thomas proves that static checking will not be affected by the addition of the cycles but that binding may become an intractable problem in some cases. His proof is based in the fact that binding requires for each node besides a name, a directory of the resources (required and provided) for each context upon which that node may be used by other modules. This list of directories may be infinite if partially recursive functions are present (cycles). Expensive dynamic linkage must be used for these cases instead. Of course Thomas obtains interconnection flexibility by going from a pure oriented tree structure of a system to an oriented tree with cycles at the price of sometimes not being able to do the binding.

The "universe of discourse" of Thomas' MIL is names which are mainly of four classes: Resources, Modules, Nodes, and Subsystems.

- Resources are the class of names within a module which can actually be made available for reference.

- Modules are units of source code (may be written in different programming languages) providing and requiring resources. The definitions of resources and modules are almost identical to the ones given in MIL75.

- Nodes are descriptive units (in MIL code) that establish environments for the modules by binding resource names to modules. Nodes are the basic entity for programming in the large just as a module description is in MIL75. So a node specifies the set of modules attached to it and the interconnection between the node and other nodes of the system.

  There are four main operations a node can apply on resources to form the MIL code:

  1. **Synthesize-** specifies a set of resources provided by a module.

  2. **Inherit-** specifies a set of resources required by a node

  3. **Generate-Locally-** specifies which modules are attached to the node being defined.

     These operators are equivalent to **provide**, **has-access-to**, and **consist-of** of MIL75 respectively.

  4. **Has-successor-** determines the set of nodes that provide resources to this node or in MIL75 terms, the successors are the children of a node that generate "derived" resources to their parent.

- Subsystems are graphs (directed) of nodes and the edges connecting them with one node (the "distinguished node") providing a characterization of the subsystem i.e. indicates resources provided and required for the whole subsystem. A subsystem is stored in a library structure and can be referenced in a MIL program as if it was a single node.

why not w/ thomas?

In [Prieto-Diaz 82] complete syntax description and examples of this MIL are presented for further reference. Figure 3-5 below shows a piece of code for this MIL to illustrate the use of the names defined above and how they describe a structure.

```
node compile
  synthesizes proc compiler
  has successors scan, parse, symtble, postfix
    successor scan
      synthesizes proc scanner
      must inherit proc symbol table
    successor parse
      synthesizes proc parser
      must inherit proc scanner
                    proc postfix
    successor symtbl
      must synthesize cluster sytable with ops enter, retrieve
      generates locally proc lookup using search
    successor postfix
      synthesizes proc postfixgen
                   proc quadruplesgen
      must inherit proc symbol table
```

Figure 3-5: Example of Code for Thomas' MIL

If a user were to design a software system using Thomas' MIL as a development tool, a structured design methodology should be followed to obtain a oriented tree structure of the system just like the "system tree" of MIL75 is obtained. The user would then define the nodes in MIL constructs by carefully analyzing which modules could be encapsulated in a subsystem so that a node structure is obtained which describes the whole system structure in a LPL. This is analogous to the "packaging" activity of structured design [Page-Jones 80]. This is a more flexible way to build the rigid "resource augmented" and "access augmented" system trees of MIL75.

During the formation of the node structure, static type checking would be performed by the MIL processor so that at the end resource flow consistency would be verified.

The next step, in contrast with MIL75, would be to code the individual modules and compile each one separately. Finally, the MIL processor would be called to do the binding and perform the required module interconnections, that

is to change all indirect references to direct connections. The MIL75 compiler instead establishes the "usage links" (bindings) at a LPL level without need for module coding.

Difference from the Other MILs

As seen in the above description, Thomas' MIL performs the module interconnection after module compilation thus allowing more flexibility to the designer at the type-check stage but at the same time forcing the complete termination of the system (coding) before interconnection can be performed. The pay-off is during maintenance when individual modules can be added without requiring full recompilation of the system as MIL75 would sometimes require. This pay-off will be incremented if the MIL were integrated into a system development environment as Thomas proposes. Thomas' MIL is restricted in two ways: 1) by using CLU-like resources and constructs and 2) by bounding the interconnection to the compile/link paradigm. Cooprider and Tichy succeed in freeing their MILs from these restrictions and in integrating their MILs in working systems development environments as will be shown below.

Experience to Date

Thomas work is only a discussion of a possible MIL processor and it was never implemented. It is however a valuable work that established certain ideas for future MILs.

3.3 Cooprider's MIL

The objective in Cooprider's work [Cooprider 79] is to propose a system that to some extent, would bridge the gap between software design and software construction. He develops a representation for software systems that integrates a MIL, a version control facility and a software construction facility. His

emphasis is on the later two facilities but succeeds in adding some innovations to the work of DeRemer & Kron and Thomas.

There are three levels of notation in this MIL. The highest, most abstract level defines the interconnection between subsystems or modules. The intermediate level describes instantiations of system versions conforming to those interconnection structures. And the lowest, most concrete level describes actual system construction operations.

The Interconnection System

The abstract portion of the subsystem interconnection notation corresponds to the one used in the previous MILs. The subsystem or module is the basic building block; resources are the currency of exchange among subsystems. Subsystems may enclose other subsystems. Resources must be named explicitly and can be "extra linguistic", that is, they are not necessarily made of programming constructs alone but may be composed of plain text or even, graphic information. All these characteristics have been defined in the previous two sections and their definition applies the same in this MIL.

There are three interconnection mechanisms in this MIL:

1. **Nesting**- The provider can be nested directly within a requirer. This mechanism is similar to the flow of resources from children to parent in the resource augmented tree of MIL75.

2. **Explicit Reference**- The provider can be named by an external clause in the requirer. This case is analogous to the accessibility channels for resources among sibling modules of MIL75.

3. **Environment Definition**- The provider can be named by a subsystem that encloses the requiring subsystem. This mechanism is the same environment described in Thomas' MIL and similar to the flow of resources from parent to children in the resource augmented tree of MIL75.

The Construction System

This lowest, most concrete level of notation is presented before the intermediate level in order to convey better understanding of the whole language. The objective here is to specify the process by which a system is constructed. concrete objects, rules, and processors are required for the construction to take place. A rule shows how a concrete object is constructed, a concrete object is a generalized file (source, object or executable code) and a processor is any program that produces a concrete object (compiler, assembler, text processor, etc..). A source file is always the original concrete object in a chain of construction rules.

There are three operators used in the construction system:

1. **file**- Used to point to a specific file name indicated by a path (full directory path) enclosed in "< >" brackets. This path may be empty thus showing the file name only.

2. **acquire**- converts a resource from another subsystem into a concrete object.

3. **deferred**- retrieves all objects that have been implicitly associated with the parameter object. This operator is used when separately compiled subroutine bodies are linked and their external procedure declarations made effective.

The example below illustrates the use of the above operators.

Example

```
concrete object file1 = FOR(file(<DIR-name:MAIN>))
concrete object COMM = FOR(acquire(COMMON-BLK))
concrete object file2 = FOR(file(source-SORT))
concrete object file3 = MERGE(file(input1),file(input2))
concrete object execMAIN = LINK(file1, file2, file3, COMM,
                                 deferred(file2))
```

The Version Control System

The objective of this system is to make different system versions share the same interconnection structure so that duplication of identical information is prevented and modification sites are centralized. This approach is better than

copying system descriptions that would require modifications to each copy for any small alteration performed to a component subsystem.

The syntax for this system consists of two parts: the realization section and the version section. The realization section contains all the information pertinent to the tangible form of a subsystem while a version is an instantiation of a subsystem or a group of such instantiations. There are several combinations of the syntactic constructs that can be used to describe a subsystem realization. The example below shows a subsystem with several versions.

Example

```
subsystem HASH provides HashFunction
 realization
  version Quick
    version Fortran resources file(<FortranQuickHash>)end Fortran
    version Pascal resources file(<PascalQuickHash>)end Pascal
    version Algol resources file(<AlgolQuickHash>)end Algol
  end Quick
  version Careful
    version Fortran resources file(<FortranCarefulHash>)endFortran
    version Pascal resources file(<PascalCarefulHash>)end Pascal
    version Algol resources file(<AlgolCarefulHash>)end Algol
  end Careful
 end HASH
```

In contrast with the two previous MILs, the language developed by Cooprider could be seen as an extended MIL that also supports system construction, not only system design. If a user were to design and construct a software system using this MIL as a development tool, a similar process would be followed as if using MIL75 or Thomas' MIL, that is, a structured design methodology. This process would be, in contrast with the previous MILs, carried on interactively with the aid of a data base where system integrity would be verified.

With this tool, construction information could also be specified and verified

during the design phase so that the end product would not be only a structured system design but also a structured description of what steps to follow to obtain such a system. Module coding could be done separately and/or in parallel with the whole system design.

The largest gain in using Cooprider's system would be by far during the evolution of the software product throughout its entire operational life.

Differences from the Other MILs

It is difficult to compare Cooprider's MIL against the previous two because of its language extensions. The module interconnection part of this tool could be considered as a synthesis of both, MIL75 and Thomas' MIL. That is, most of their advantages were integrated in this MIL such as flexibility in the interconnection structure, easy syntax and notation, and static binding. The flow of resources however, has similar restrictions as in MIL75 but not as stringent. A subsystem here only provides and requires resources in a way similar to scope rules in structured programming languages while in a module in MIL75, derived and accessed resources must also be specified depending if they flow among parent-offspring or among siblings respectively. This reduction in the complexity of resource flow is due to the use of a data base processor instead of a compiler. This is the major contribution of this MIL. The data base processor is also a key factor for the implementation of the construction and version control systems.

A drawback of the construction mechanism is that the data base has no knowledge of the nature of the various versions. Therefore the realization description requires excessive detail and the designer must give explicit construction rules for all components and configurations as well as program all

the modification policies by hand. Moreover the data base processor does not support control for concurrent actions (i.e. two programmers modifying the same file at the same time.)

Experience to Date

Several parts of this system have been implemented. The implemented components were tested in a laboratory environment with a specific and small test case: A software support for a scan line graphics printer. They have not been proved in a real production environment. There is no report of a consistent version of the system as proposed but many of the ideas and some of the components have been used in the development of the System Generator Facility of the GANDALF System [Haberman, et.al. 81].

3.4 INTERCOL

The goal of Tichy's work at the software development environment level envisions three objectives:

1. A Module Interconnection Language (INTERCOL) capable of representing multiple versions and configurations written in multiple programming languages.

2. An Interface Control System that automatically verifies interface consistency among separately developed software components.

3. A Version Control System similar to the one proposed by Cooprider [Cooprider 79] but with the advantage that in this case the system determines which version of which component should be combined to form a particular version of a particular configuration instead of relying on a detailed set of construction commands issued by the designer as in Cooprider's MIL.

A description in INTERCOL is a sequence of module and system families followed by a set of compositions. A member of a module family is a version of a module, and a member of a system family is a version of a system. The former may be one of a set of different module implementations for different environments

or in different languages, or may be one of a set of different module revisions, or can also be a derived version. The latter may be a member of a set of different system configurations or of a different derived composition.

Each one of the above families has an interface. An interface consists of programmed entities called resources. A resource in INTERCOL has the same meaning as a resource in the previous MILs; they are the units of flow among modules and/or among systems. All members of a particular module or system family use the same interface so that free substitution of family members can occur. This is the main reason, in contrast with previous MILs, that INTERCOL makes every interface explicit.

INTERCOL interacts with a number of different programming languages by means of a resource-specification sublanguage. Resources are constructs in a specific programming language that are implemented and used in the modules. Thus a mapping from resource specification sublanguage is installation dependent, but the language must be statically typed. The sublanguage used by Tichy in his work is a subset of the Ada language.

A resource declaration in INTERCOL may consist of a compact representation or a specification or both. A compact representation is an abbreviated list of resources and their attributes (type, access, etc.) and a specification is a list of resources written in the resource specification sublanguage.

A module family has an interface consisting of a list of provided and required resources and contains one or more implementations. Each implementation may exist in several revisions which are the entities or files that contain the actual programs. Different programming languages can be used for different

realizations. Each realization may have several revisions, where a revision is the result of programming the initial revision or editing an existing one. Derived versions constitute a second dimension of variation of realizations.

A system family contains zero or more module and system families and zero or more compositions. A composition gives a name to a combination of elements that are the names of previously declared building blocks in the same or enclosing system families.

The construction process of a software system followed by a user of INTERCOL would be almost identical to the process described for a user of Cooprider's MIL. INTERCOL however, is imbedded in a "Software Development Control Facility" (SDCF) which is organized as an interactive system that controls a software development data base. SDCF moreover, allows for separate and incremental (asynchronous) compilation of modules, and independent type checking thus significantly reducing development costs.

The advantage of using Tichy's SDCF over the previous MILs is at the level of controling the evolutionary process of a software system. The approach of system design by "evolving prototypes" would be the ideal approach to use with this SDCF.

Differences from the Other MILs

The most significant contributions of INTERCOL and Tichy's SDCF to MILs are:

1. Allows a structured specification and control of families of systems which enclose families of modules.
2. Allows separate and asynchronous compilations of modules and independent type checking.
3. Includes an interface control system that automatically manages the consistency of the interconnection among module and system families.

4. Includes a version control system that supervise the addition of new versions.

Experience to Date

Tichy's SDCF is operational at the prototype level in a PDP 11/40 system under UNIX and has been integrated into the GANDALF System [Haberman, et.al. 81] as the System Version Description facility. There has been no reports of the SDCF being used in a real software development project. As of 1981 GANDALF had not yet been used in a software production environment. There is no test data of Tichy's work to evaluate its performance and effectiveness. What has been proved however, is its feasibility.

4. SYSTEMS SUPPORTING MODULE INTERCONNECTION

There are several programming environments and software development systems that provide module interconnection facilities. Detailed classification of these systems is difficult (and beyond the scope of this paper) since each has its own unique characteristics and they emerged from diverse design philosophies.

From the point of view of module interconnection however, we have selected a representative sample of the different approaches taken to perform module interconnection. The sample includes the following systems: PWB, CLU, ADAPT, MESA, PROTEL, CDL2, SARA and, GANDALF. Each of these approaches will be discussed in more detail in the following sections.

need a reference to PWB

PWB represents the class of systems that provide facilities for management of system development (i.e. version control) but lack facilities for strict module interconnection (i.e. intramodule type-checking, systems structure description, module accessibility). Systems like the Software Factory [Bratman & Court 75], the SWB System [Matsumoto 81] and, the ARCTURUS System [Standish 81] also fall in this class.

CLU and ADAPT represent the class of languages and language extensions perfectly suited to support module interconnection. Languages in this class are highly modular and provide constructs for version definitions. They are based on data abstractions and use the same language for module construction (PS) as for system description (PL). This last characteristic is a drawback from the point of view of MILs. MODULA and ALPHARD among others, also fall in this class.

Should say this much earlier

MESA, PROTEL and, CDL2 represent the class of fully integrated software development systems that are actually used in production environments. This

class of systems perform module interconnection as MILs do but are restricted to modules written in their own languages thus inhibiting the use of modules written in different programming languages.

SARA represents the class of special purpose software design systems at the development stage that use different approach to module interconnection (SARA advocates the formal model approach to MILs).

GANDALF represents the class of fully integrated software development systems that have successfully included one of the MILs described above (INTERCOL) as one of their development tools.

4.1 PWB

The PWB (Programmer's WorkBench) facility provides limited support for module interconnection. Based in UNIX, PWB was developed by Bell Labs in 1973 ([Dolotta & Mashey 76], [Ivie 77], and [Bianchi & Wood 76]) to provide tools and services to ease the load on the application system designer, programmer, documenter, tester, and development personnel. It is based on the concept that the facilities needed by program developers bare some difference from those required by the program users.

PWB succeeds in separating the program development and maintenance function onto a specialized computer which is dedicated to that purpose. This computer provides the interface between program developers and their target computer(s). PWB supplies a separate uniform environment in which people perform their work. The facilities supported by the PWB (as of 1979) are a source control system, a remote job entry system, a document preparation system, a modification request control system, and drivers that simulate user conditions for testing.

The PWB Source Code Control System (SCCS) [Rochkind 75], is a file storage system that records the various versions of a text file; this is accomplished by recording the original version plus interleaved modification descriptions that can be applied to create more up-to-date versions. This system provides: creation of any revision of a source program or text, file protection against accidental changes, selective propagation of module changes to each of its revisions and, identification of object and source (revision number, date created, etc.).

PWB's SCCS System does not provide syntax constructs for module interconnection descriptions as a MIL would do. SCCS is a version control tool only.

4.2 CLU

The programming language (CLU) was designed by Liskov [Liskov et.al. 77] to implement the concept of abstract data types. It provides constructs that support the use of abstractions in program design and implementation. A similar language (ALPHARD) [Wulf 74] was designed mainly to support the construction of structured programs. Both deal with abstract data types and abstraction building mechanisms. Both are derived from SIMULA 67 ([Dahl, Myrhaug & Nygaard 70] and [Birtwistle et.al. 73]). Although CLU and ALPHARD are somewhat similar, they differ in many important details.

In CLU, programs are developed incrementally, one abstraction at a time. A distinction is made between an abstraction and a program or module which implements that abstraction. An abstraction isolates use from implementation: "An abstraction can be used without knowledge of its implementation and implemented without knowledge of its use." The CLU library which supports this

methodology, maintains information about abstractions and the CLU modules that implement them.

For each abstraction there is a description unit which contains all system-maintained information about that abstraction. The interface specification which is that information needed to type-check uses of the abstraction is the most important information of an abstraction contained in a description unit. In most cases, this information consists of the number and types of parameters, arguments, and output values plus any constrains on type parameters.

An abstraction is entered in the library by submitting the interface specification; no implementations are required. A module can be compiled before any implementations have been provided for the abstraction it uses. During compilation the external references of a module must be bound to description units so that type checking can be performed. The binding is accomplished by constructing an association list, mapping names to description units, which is passed to the compiler along with the source code when compiling the module. The mapping in the association list is then stored by the compiler in the library as part of the module.

The idea of compiling the abstractions with their interface specifications without any implementations needed is the very same idea of MIL75. An important feature of CLU is its type checking capability across modules, which is a natural consequence of its objective: to aid the programmer to construct correct programs. A drawback is its lack of support for system organization.

Cooprider showed that a MIL based on a data base processor is more effective in the control of system organization than a MIL based on a compiler. It could

be argued then, that the CLU library is the equivalent of a data base processor because it supports incremental program development but can not however, support version nor system family control because the compiler binds a module permanently to the abstractions it uses. This is the price of strong type-checking needed for correct programs. CLU therefore is more of a LPS (Language for Programming in the Small) than a LPL.

Reference the recent IBM S. Journal article

4.3 ADAPT

ADAPT (Abstract Design And Programming Translator), a language resembling CLU in its essentials but with PL/1-style syntax, has been implemented at IBM [Archibald 81]. It has proven to be as good a mechanism for describing the detailed semantics of modules as CLU is but, in contrast with CLU, a MIL has been added. This MIL extension to ADAPT is called External Structure.

The External Structure is a MIL used primarily for system description with a facility to convert the syntax description into a graphic display. It is used as a design tool and as a project control facility that provides system structuring support for programmers and development groups. It allows for separate compilation of modules and performs inter-module type checking. It is an automated resource, interacting with the ADAPT compiler.

A drawback in the module interconnection mechanism of the External Structure is the restriction to modules written in the ADAPT language. A system is described in External Structure as a collection of modules and their allowable interconnections. This approach is very similar to the one followed by C/MESA of the MESA System.

4.4 MESA

MESA was developed by XEROX at XEROX PARC during 1975 ([Geschke et.al. 77] and [Mitchell, et.al. 79]) and is successfully being used in the design, specification and implementation of a number of systems. In particular, the experience of using MESA for the development of a operating system is reported in [Lauer & Satterthwaite 79] and [Horsley & Lynch 79].

In contrast with the MILs described in the previous sections, MESA is both a programming language and a software development system, and it is currently being used in a production environment. MESA supports program modularity as the basis for incremental program development and provides complete type checking for subsystems to be developed separately and safely bound together. The MESA language is similar to Pascal or Algol 68 and with a global structure similar to that of Simula. MESA by itself would be a strongly typed LPS supporting separate compilation but, with the addition of C/MESA which provides separate configuration descriptions, it became a very powerful and practical MIL.

C/MESA, a configuration language developed in 1978, describes the organization of a system and controls the scope of interfaces. C/MESA has many of the attributes of a MIL as described in section 2 above and is used in the MESA system to specify how separately compiled modules are to be bound together to form configurations.

From the MILs point of view, MESA and C/MESA form a well integrated set of tools covering the design and implementation aspect of the life-cycle of a software system. The MESA System succeeds in implementing some of the ideas originated in MIL75 and parallels some of the ideas of Cooprider and Tichy on version control but at a less general level. The goal of C/MESA is to allow the

user to represent a complete system in a hierarchy of configuration descriptions. In MIL75 terms, C/MESA has all the syntactic constructs to represent a system tree.

Systems built in MESA are collection of modules of two kinds: definitions and programs. A definitions module defines the interface to an abstraction by declaring shared types and constants and by naming procedures available to other modules. Program modules are pieces of source text similar to Algol procedure declarations or Simula class definitions.

A module declaration in MESA defines a data structure consisting of a collection of variables and a set of procedures that operate on those variables. This concept of a module is more restricted than that used by the MILs described above because at the level of module definition MESA is a programming language only. Modules communicate with each other via interfaces. A module may import an interface, in which case it may "use" facilities defined in the interface and implemented in other modules. The importer is called a client of the interface. A module may also export an interface, in which case it makes its own facilities available (provides) to other modules as defined by that interface. Such a module is called implementor.

An interface consists of a sequence of declarations defined by a definitions module. Only the names and types of operations are specified in the interface, not their implementations. Figure 4-1 below illustrates a definitions module and one of its implementors.Modules and interfaces are compiled separately. The compiler reads each of the imported modules and obtains all of the information necessary to compile the importing module performing type-checking for all references. No knowledge about any implementors of the interfaces is required.

```
Abstraction:DEFINITIONS =
  BEGIN
  ....
  it:TYPE=....;rt:TYPE=....;
  ....
  p:PROCEDURE;
  pt:PROCEDURE[it] RETURNS[rt];
  ....
  END


Implementer:PROGRAM IMPLEMENTING Abstraction =
  BEGIN
  OPEN Abstraction;
  x:INTEGER;
  ....
  p:PUBLIC PROCEDURE = <code for p>;
  p1:PUBLIC PROCEDURE[i:INTEGER] = <code for p1>;
  ....
  pi:PUBLIC PROCEDURE[x:it] RETURNS[y:rt] = <code for pi>;
  ....
  END
```

Figure 4-1: A Definitions Module and an Implementor in MESA Taken from [Geschke et.al. 77]

The MESA binder collects exported interface records which have been identified with a unique name by the compiler, and assigns their values to their corresponding interface records of the importers. This unique name is what allows the binder to check that each interface is used in the same version by every importer and exporter. The binder uses the configuration description program (coded in C/MESA) to bind modules together to form configurations. Figure 4-2 below shows the partial code for a system configuration. In this example, A, B, C,... are the interfaces and U, V, W,... are the modules that import/export them as indicated by the special comment characters (—).

The definitions modules of MESA are equivalent to the declarative statements of any of the MILs described above and the separate C/MESA code is equivalent to a MIL program without the declarative statements. For example, a definitions

```
Config1:CONFIGURATION
            IMPORTS A
            EXPORTS B =
BEGIN
    U;                  --imports A,C
    V;                  --exports B,C
END.

Config2:CONFIGURATION
            IMPORTS B =
BEGIN
    W;                  --imports B, Exports C
    X;                  --imports B,C
END.

Config3:CONFIGURATION
            IMPORTS A =
BEGIN
    Config1;
    Config2;
END.
```

Figure 4-2: A Partial Configuration Description in C/MESA Taken from [Geschke et.al. 77]

module in MESA has statements analogous to **provides**, **originates**, and **consist of** from MIL75 and to **synthesizes**, **inherit**, and **has successors** from Thomas' MIL. Such statements in MESA however, are not explicit as in the MILs but rather implicit as observed in the example of figure 4-1.

The separate C/MESA code as illustrated by the example of figure 4.7, explicitly uses **IMPORTS** and **EXPORTS** predicates to define resource flow but does not give an explicit view of the resources imported and exported by each of the component modules. Such declarations are implicit in each module and the C/MESA programmer must make such declarations visible with comments.

This approach to module interconnection is different from the approach advocated by the MILs described above. The module interconnection facility offered by the MESA System is a combination of an implicit declaration of resource flow by each module and an explicit configuration description. In

contrast, the other MILs propose a separate module description and system configuration coding where all resource flow is explicit. A drawback however, is that MESA modules are restricted to the MESA language.

In contrast with the other MILs, MESA is a widely used and tested facility within XEROX where a substantial amount of experience on its use has been accumulated.

4.5 PROTEL

PROTEL (PRocedure Oriented Type Enforcing Language) is a tool that supports type checking across modules in a fashion similar to MESA [Cashin, et.al. 81]. PROTEL was implemented in 1975 by Bell-Northern Research of Ottawa, Canada and has been used extensively since then mainly by its own developers.

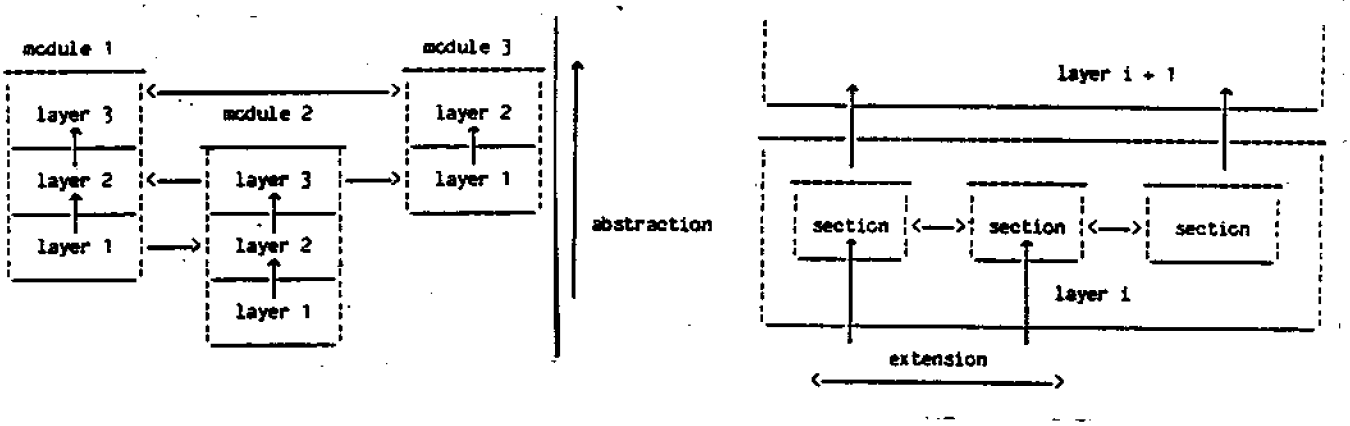
This system is based on the compile-link-load paradigm like UNIX but performs type checking across modules like MESA. To support type checking of inter-section and inter-module references, the compiler performs a process called embedding which consists of first writing symbolic information to an object file and then reading that information for all sections visible to the one being compiled and using it to initialize the compiler symbol table. With the symbol table so initialized, full compile time checking of all references can take place.

A Library System was added in 1977 to support module interconnection and system version control but resulted in an environment too involved to be practical [Cashin, et.al. 81]. PROTEL is very limited in controlling system versions and in supporting system organization. The Library System is restricted to modules coded in PROTEL.

4.6 CDL2

The CDL2 system is a development environment centered around one single language, the implementation language CDL2 [Bayer 81]. It is intended for the development of large, sequential and non-numeric systems. The CDL2 lab was developed at the Technical University of Berlin between 1977 and 1980.

From the MILs point of view, a CDL2 program consists of modules, which may be connected via explicit export and import interfaces. Each module contains a strict hierarchy of layers. Lower layers can export resources to layers at a higher level of abstraction. In figure 4-3(a) horizontal arrows represent import/export module interfaces and upper pointing arrows show layer export direction.



a: Module Level b: Layer Level

Figure 4-3: Modular Structure in CDL2 from [Bayer 81]

Within the layers is found a set of sections connected by explicit interfaces to other sections in the same layer or in the next higher. Export of resources within one same layer is called extension whereas exports to the next higher layer are called abstractions. Sections are functional units like modules are.

Extension is used to extend the power of a layer, abstraction is used to realize one level of the abstract machine in terms of the next lower one. The brakedown of modules into layers and of layers into sections allows the user to define different layers of interconnection. If a module for example, is originally designed at a very low level of abstraction, a higher level description of the same module could be designed to provide a simpler, more general interconnection interface.

The unique hierarchical organization of modules in CDL2 provides the structure for describing a very high level design that could accommodate different versions of the same module. The lower level layers of a particular module could be replaced by layers performing the same function but having different characteristics. This property is very effective when constructing transportable systems.

The CDL2 System is centered around a command interpreter that gives the user a uniform language to control all components of the system (Editor, Formater, Analyzers, Coders and, Database). From the MILs point of view, the Local and Global analyzers are essential because together they perform the role of a MIL. The Local Analyzer consists of a Syntax Checker and a Local Semantic Checker. The Global Analyzer consists of a Global Semantic Checker and an Intermodule Interface Checker. The Itermodule Interface Checker is used during system design and specification to create a general design description. The Global Semantic Checker verifies import/export data types across modules (similar to the MIL75 compiler). The Local Semantic Checker verifies internal module interfaces (among layers).

The CDL2 System is presently being used in various research projects within

the Technical University of Berlin and has been transported to other sites where it is being used as an experimental software development environment.

4.7 SARA

SARA (System ARchitect's Apprentice) is a computer-aided system which supports a structured multi-level requirement driven methodology for the design of reliable software or hardware digital systems. SARA was designed at UCLA in 1976 ([Estrin 78], [Campos & Estrin 78a], and [Campos & Estrin 78b]) and has been under continual development since then.

The SARA methodology, based on formal models, supports both a top-down partitioning procedure (refinement) and a bottom-up composition procedure (abstraction). It deals mainly with the structure of the record of execution providing effective means for synthesizing and analyzing a system. To accomplish this, SARA makes use of a structural model (SL1) and a behavioral model (GMB - Graph Model of Behavior).

The structural model resembles the contour model [Johnston 71] used to describe the semantics of algorithm execution in block structured processes. The contour model consists of graphs that represent processes enclosing nested blocks. The structural model consists also of enclosing contours but in this case they are used mainly to enforce modularity by providing a better means to enforce encapsulation. They permit the isolation of parts of the system which then can be modeled separately. SL1 is SARA's modeling language designed to describe the structure of a modular system.

The behavioral model consists of two graphs: a flow-of-control (CG - Control Graph) and a flow-of-data (DG - Data Graph) together with interrelations associated with the nodes of the data graph. The CG is a Petri-net of processes

and directed control arcs and the data flow is modeled in the DG through processors and data sets, where the processors are responsible for the transformation of the data stored in data sets.

The structure of the record of execution is effectively accomplished by mapping between the behavioral and structural models. This mapping provides the SARA tools with means to detect any inconsistency in the design but, it does not provide any facilities for module interconnection, leaving SARA as a methodology for system specification and design but not for system implementation.

In 1979 a MIL was added to SARA [Penedo & Berry 79] to deal with the algorithm structure. This MID (Module Interface Description)[1] is intended to enhance the power of SARA by providing a smoother path from modeling to code.

In this new model a SARA-MID mapping is obtained in which the SLI-GMB model identifies the variables and the calls of the code; the MIL model identifies the type and procedure definitions; and the mapping (SARA-MID) says which variable is of what type and which call is of what procedure.

As of 1979 the SARA-MID methodology was only a model open to many questions about efficiency, effectiveness, and performance. Work is still being conducted on its integration into the SARA system.

How old is this paper? Should update it.

---

[1]This work was formerly called MISC for <Module Interconnection Specification Capability

4.8 GANDALF

GANDALF [Haberman, et.al. 81] is a new software development environment to some extent different from all the conventional tools, such as the ones described above. It is designed for projects that use the new Ada language ~~and~~ but its current implementation is written in the C language.

It is called an "environment" rather than a "tool" because it integrates uniformly a set of three development support tools. These tools can cooperate closely with each other since they are all based on Ada and are generally knowledgeable about the environment. They operate on a common representation: the syntax tree representation of the program. These three development support tools are:

1. A collection of incremental program construction tools
2. A collection of system version description and generation tools, and
3. A collection of project management tools.

The incremental program description tool consists of a syntax directed editor and a syntax directed dynamic debugger. The syntax directed editor is formed by the pair (program constructor, unparser) as a replacement for the typical triple (line editor, lexical analyzer, syntax analyzer). This new approach allows the programmer to write syntactically correct programs the first time around.

The idea of the dynamic debugger is that a user can write his debuging statements in terms of the source representation of his program instead of in terms of machine code, memory locations and fast registers. A program can be built incrementally because the program or subprogram being debuged is halted, corrected, recompiled, linked, and loaded automatically. Execution can then be continued upon modification.

The System Version Description and Generation Tool is actually a MIL developed by Cooprider and later by Tichy. This MIL addresses the two basic problems of system composition: module interface control and system version control (a more detailed description is given in the next section). It provides a system generation facility based on system descriptions thus taking over all necessary bookkeeping from programmers or system builders, qualitative improvement over UNIX, MESA, PROTEL, and SARA. Type checking across modules and system boundaries is also provided and performed independently and/or incrementally thus helping the system builder in assembling perfectly matched modules.

The purpose of the Project Management facility is to support collaboration of programmers on a project. It consists of two parts: 1) Software Development Control (SDC), responsible for coordinating the state of the system, and 2) Generation and proliferation of documentation.

The former is also responsible for avoiding conflicts of interest among project programmers; i.e. it will not permit two programmers to alter a source concurrently. Access rights are automatically checked by the system so that unauthorized users may not manipulate the project.

The later part is still under development. It is intended to force users to comment on source object manipulations by prompting programmers for documentation whenever additions or modifications are made to the system. This ensures that there is no time when a change is been made to the system state that is not reflected in the documentation.

In contrast with other systems composed of tools that are used individually

for different tasks, GANDALF provides a well integrated environment that uses among other tools the latest MIL for module and version control. GANDALF may be considered as one of the first revolutionary software development environment of the 80's. It is built on most of the ideas described in the previous sections. It uses, for example, the concept of structured programming and stepwise refinement for construction of modular programs; the ideas of Parnas [Parnas 72] for module construction using information hiding; the concept of separating system specification (LPL) from implementation (LPS) [DeRemer & Kron 76]; system version representation by abstract data types; and several other ideas from previous tools i.e. UNIX, MESA, CLU, etc.. GANDALF has been implemented and is currently under evaluation.

4.9 TOOL SUPPORT FOR INTERCONNECTION

Figure 4-4 below illustrates graphically the relationship among the tools described above and their support of some kind of module interconnection.

Full Support Tools:

Gandalf	C/Mesa	SARA-MID	CDL2
Intercol	Cooprider's MIL	Thomas' MIL	MIL75

Partial Support Tools:

Make	Protel	Ada	Mesa
PWB	NOPAL	SARA	

Marginal Support Tools:

CLU	SIMULA	MODEL	ALPHARD
modular languages			

Figure 4-4: Some Tools Supporting Module Interconnection

5. CONCLUSION

After taking the reader through this long and detailed description of module interconnection languages and of software development systems that support some kind of module interconnection, it would be worthwhile to mention at least their main contributions to the partial solution of the present "software crisis".

MILs and their related processors represent a set of tools which primarily aid the software engineer during the architectural design, evolution and maintenance phases of the system life-cycle. A secondary purpose of MILs is to serve as a goal for systems analysis and a constraint for systems implementation. To be effective, a MIL must be integrated into a software development system or facility where the MIL description of a system is checked every time a change to that system is made.

Within this range of effectiveness of the MILs, the main contributions are:

1. MILs provide a means to represent the architectural design of a software system in a separate machine checkable language. Design and construction information is successfully integrated at the programming-in-the-large level. These notations should be of interest to researchers in automatic programming and program generation since they are developing mechanisms to manipulate this information.
2. MILs can prohibit programmers from changing the system architectural design during evolution and maintenance without an explicit change in the architectural design as represented by the MIL.
3. MILs can represent the construction process for a system and can serve as the basis for a unified data base during system development.
4. A consequence of (1) is a substantial improvement of the maintenance stage. A system can be revised, modified and type checked at the MIL level before attempting any changes to the code.

These contributions although significant, are only a small step towards the solution of the software crisis. On the other hand, some of the main limitations of MILs can also be listed.

1. The contribution of MILs to the design stage is mainly in checking design completeness, not in performing the design. The design must be carried out by means of the present methodologies or techniques.

2. A MIL becomes an effective tool only in very large systems. The amount of effort required to use a MIL along with the development of a system is very large and it pays-off only if maintenance is extensive.

3. MILs do not provide any means for the user to determine which of the already constructed modules can be used when designing a new system. This problem of course was not intended to be solved by MILs, but seems to be a very attractive feature to have considering the information contained in a MIL description of a system.

Bad sentence

A question naturally comes to mind: To what extent could the main ideas and concepts of MILs be used to improve other stages of the software life-cycle?

6. FUTURE RESEARCH

Some of the main concepts of MILs could be used as driving ideas in other areas of current research in computer science in general and in software engineering in particular.

The idea in MILs of a separate language to describe system structure could be extended to study the problem of representing system specifications. A "module specification language" could be proposed together with a study of what methods we must develop for encoding general specifications and how could we match requirement specifications with provision specifications. Among the issues to be addressed with this proposition are compatibility, upward compatibility, functional equivalence, minimal satisfaction, uniformity, and type [Cooprider 79]. Reusability is also an important issue directly related to this matching scheme.

Reusability, as proposed by Freeman [Freeman 80] and Neighbors [Neighbors 80], should seldom deal with executable code and primarily use non-executable work products from system analysis and design. A research question is then how could a MIL be expanded or augmented to include information about availability of resources and modules? At present, MILs provide a description of system structure and resource flow among modules (system components) but more information is needed to indicate the specifications of such modules and resources. How much information is needed to be able to decide whether this or that module will satisfy the proposed design requirements?.

Program generation techniques is an area where some MIL concepts have been used. MODEL [Prywes 77] and NOPAL [Sangal 80] are two non-procedural languages used for automatic generation of computer programs that support module

description and provide limited module interconnection. There is however, a need of extensive research in this area. The way MILs consolidate design and construction processes in a single description for example, could provide some insight into the question of encoding the methods by which information from a problem is encoded in programs.

There are further research questions that relate both the reusability problem and the automatic program generation problem. The following question touches the very concept of reusability. To what extent is it practical to reuse components that can be easily generated by automatic programming systems?. Maybe it would be more practical to reuse construction processes as represented in MILs than to reuse design specifications (the first being a high level executable code, the second a non-executable work product). To reuse a construction process would be however, more attractive than reusing a design specification.

Another area that deserves research was proposed by Tichy [Tichy 80]. He suggests the study of techniques to implement automatic retesting after changes to insure that an error that has been found previously, has not been re-introduced.

A common symptom of large and successful systems is massive change over a long period of time. These changes occur along three lines: evolution (system functional change), maintenance (system error correction), and hardware/software changes (configurations) supported by the system. Each of these changes provides an index into a "version space" for a particular system. The MILs of Cooprider [Cooprider 79] and Tichy [Tichy 80] started to examine the problem of version control but much more work is needed. The problem of which changes a

new version of a system along some dimension inherits from the other dimensions remains unsolved.

In conclusion, having examined most of the existing MILs, some of the software development tools that support module interconnection, and their significant contributions to improving the state of the art in software engineering technology we find out that there is still a long way to go before a major breakthrough in the manufacture of software is achieved. Every major breakthrough in technology however, has been attained through small steps.

ACKNOWLEDGMENTS

We are deeply indebted to Peter Freeman for his untiring support and encouragement, as well as for introducing the authors to MILs. His valuable comments while reading the first drafts of this work are appreciated. We want to thank Anthony Wasserman for detailed and constructive comments as well as Haydée Prieto for her editorial assistance.

REFERENCES

[Archibald 81]
Archibald, J.L.
The External Structure: Experience with an Automated Module Interconnection Language.
The Journal of Systems and Software, 2(2):147-157, June, 1981.

[Balzer, Goldman & Wile 76]
Balzer, R.M., Goldman, N.M., and Wile, D.
On the Transformational Implementation Approach to Programming.
In Proceedings of the Second Intl. Conference on Software Engineering, pages 337-344. IEEE, 1976.

[Bayer 81]
Bayer,M. et.al.
Software Development in the CDL2 Laboratory.
In Software Engineering Environments, pages 97-118. North-Holland, 1981.

[Bianchi & Wood 76]
Bianchi, M.H. and Wood, J.L.
A User,s Viewpoint on the Programmer's Workbench.
In Proceedings of the Second Intl. Conference on Software Engineering , pages 193-199. IEEE, October, 1976.

[Birtwistle et.al. 73]
Birtwistle, G.M., Dahl, O-J., Myhrhaug, B., and Nygaard, K.
Simula BEGIN.
Petrocelli/Charter, 1973.

[Bratman & Court 75]
Bratman, H., and Court, T.
The Software Factory.
IEEE Computer, 8(5):28-37, May, 1975.

[Campos & Estrin 78a]
Campos, I.M., and Estrin, G.
SARA Aided Design of Software for Concurrent Systems.
In Proceedings of the National Computer Conference. AFIPS Press, 1978.

[Campos & Estrin 78b]
Campos, I.M., and Estrin, G.
Concurrent Software System Design Supported by SARA at the Age of One.
In Proceedings of the Third Intl. Conference on Software Engineering, pages 230-242. IEEE Press, Atlanta, Georgia, USA, May, 1978.

[Cashin, et.al. 81]
Cashin, P.M., Joliat, M.L., Kamel, R.F., and Lasker, D.M.
Experience with a Modular Typed Language: PROTEL.
In Proceedings of the Fifth Intl. Conference on Software Engineering, pages 136-143. IEEE, San Diego, California, March, 1981.

[Cooprider 79]
Cooprider, L.W.
The Representation of Famlies of Software Systems.
PhD thesis, Carnegie-Mellon University, Computer Science Department, April, 1979.
CMU-CS-79-116.

[Dahl, Myrhaug & Nygaard 70]
Dahl, O.J., Myrhaug, B. and Nygaard, K.
The SIMULA 67 Common Base Language.
Technical Report S-22, Norweigan Computing Center, 1970.

[DeRemer & Kron 76]
DeRemer, F., and Kron, H.
Programming-in-the-Large Versus Programming-in-the-Small.
IEEE Transactions On Software Engineering, June, 1976.
This paper was presented at the International Conference on Reliable Software, Los Angeles, California, April 1975.

[Dijkstra 65]
Dijkstra, E.
Programming Considered as a Human Activity.
In Proceedings of the 1965 IFIP Congress, pages 213-217. North Holland Publishing Co., Amsterdam, The Netherlands, 1965.

[Dolotta & Mashey 76]
Dolotta, T.A. and Mashey, J.R.
An Introduction to the Programmer's Workbench.
In Proceedings of the Second Intl. Conference on Software Engineering, pages 164-168. IEEE, October, 1976.

[Estrin 78]
Estrin, G.
A Methodology for Design of Digital Systems - Supported by SARA at the Age of One.
In Proceedings of the National Computer Conference. AFIPS Press, 1978.
Vo. 47.

[Freeman 80]
Freeman, P.
Reusable Software Engineering: A Statement of Long-Range Research Objectives.
Technical Report TR 159, University of California, Irvine, November, 1980.

[Geschke et.al. 77]
Geschke, C.M., Morris, J.H., and Satterthwaite, E.H.
Early Experience with MESA.
Communications of the ACM, 20(8):540-552, August, 1977.

[Gries 81]
Gries, D.
Texts and Monographs in Computer Science. : The Science of Programming.
Springer Verlag, New York, 1981.

[Haberman, et.al. 81]
Haberman, N., Perry, D., Feiler, P., Medina-Mora, R., Notkin, D., Kaiser, G., and Denny, B.
A Compendium of Gandalf Documentation.
Carnegie-Mellon University, Pittsburg, Pennsylvania, 1981.

[Horsley & Lynch 79]
Horsley, T.R. and Lynch, W.C.
Pilot: A Software Engineering Case Study.
In Proceedings of the Fourth Intl. Conference on Software Engineering, pages 94-99. IEEE Press, Munich, Germany, September, 1979.

[Ivie 77]
Ivie, E.L.
The Programmer's Workbench - A Machine for Software Development.
Communications of the ACM, 20(10):746-753, October, 1977.

[Johnston 71]
Johnston, J.
The Contour Model of Block Structured Processes.
In SIGPLAN Notices-Proc. Symp. Data Structures and Prog. Languages, pages 55-82. ACM, 1971.

[Lauer & Satterthwaite 79]
Lauer, H.C., and Satterthwaite, E.H.
The Impact of MESA on System Design.
In Proceedings of the Fourth Intl. Conference on Software Engineering , pages 174-182. IEEE, Munich, Germany, September, 1979.

[Liskov et.al. 77]
Liskov, B., Snyder, A., Atkinson, R., and Shaffrt, C.
Abstraction Mechanisms in CLU.
Communications of the ACM, 20(8):564-574, August, 1977.

[Matsumoto 81]
Matsumoto, Y. et.al.
SWB System: A Software Factory.
In Software Engineering Environments, pages 305-318. North-Holland, 1981.

[Mitchell, et.al. 79]
Mitchell, J.G., Maybury, W., and Sweet, R.E.
Mesa Language Manual.
Technical Report CSL-79-3, Xerox Corp., Palo Alto Research Center, April, 1979.

[Neighbors 80]
Neighbors, J.M.
Software Construction Using Components.
PhD thesis, University of California, Irvine, 1980.
ICS Technical Report 160.

[Newell et.al. 61]
Newell, A., Tonge, F.M., Feigenbaum, E.A., Green, B.F., Mealy, G.H.
Information Processing Language-V Manual
Second edition, The RAND Corp., Englewood Cliffs, N.J., 1961.
printed by Prentice-Hall, Inc.

[Page-Jones 80]
Page-Jones, M.
The Practical Guide to Structured Systems Design.
Yourdon Press, 1980.

[Parnas 72]
Parnas, D.L.
On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the ACM, 15(12):1053-1058, December, 1972.

[Penedo & Berry 79]
Penedo, M.H., and Berry, D.M.
The Use of a Module Interconnection Language in the SARA System Design Methodology.
In Proceedings of the Fourth Intl. Conference on Software Engineering, pages 294-307. IEEE Press, 1979.

[Prieto-Diaz 82]
Prieto-Diaz,R. and Neighbors, J.M.
Module Interconnection Languages: A Survey.
Technical Report UCI-ICS-TR189, University of California, 1982.

[Prywes 77]
Prywes, N.S.
Automatic Generation of Computer Programs.
In Advances in Computers. Academic Press, 1977.

[Rich, Schrobe & Waters 79]
Rich, C., Schrobe, H.E., and Waters, R.C.
Overview of the Programmer's Apprentice.
In Proceedings of the Sixth Joint Conference on Artificial Intelligence, pages 827-828. Stanford Computer Science Dept., 1979.

[Rochkind 75]
Rochkind, M.J.
The Source Code Control System.
IEEE Transactions on Software Engineering, se-1(4):364-370, December, 1975.

[Sangal 80]
Sangal, R.
Modularity in Non-Procedural Languages Through Abstract Data Types.
PhD thesis, The Moore School of Electrical Engineering, University of Pennsylvania, August, 1980.

[Standish 81]
Standish, T.A.
ARCTURUS An Advanced Highly-Integrated Programming Environment.
In Software Engineering Environments, pages 49-60. North-Holland, 1981.

[Stevens, et.al 74]
Stevens, W.P., Meyers, G.J., and Constantine, L.L.
Structured Design.
IBM Systems Journal, 1974.

[Thomas 76]
Thomas, J.W.
Module Interconnection in Programming Systems Supporting Abstraction.
PhD thesis, University of Utah, June, 1976.

[Tichy 79]
Tichy, W.F.
Software Development Control Based on Module Interconnection.
In Proceedings of the Fourth Intl. Conference on Software Engineering, pages 29-41. IEEE Press, September, 1979.

[Tichy 80]
Tichy, W.F.
Software Development Control Based on System Structure Description.
PhD thesis, Carnegie-Mellon University, Computer Science Department, January, 1980.

[Wulf 74]
Wulf, W.A.
ALPHARD: Toward a Language to Support Structured Programs.
Technical Report, Carnegie-Mellon University, Computer Science Department, April, 1974.

[Yourdon & Constantine 79]
Yourdon, E. and Constantine, L.L.
Structured Design: Fundamentals of a Dicipline of Computer Program and Systems Design.
Prentice-Hall, Englewood Cliffs, N.J., 1979.