MODULE INTERCONNECTION LANGUAGES

A SURVEY

TR#189

RUBEN PRIETO-DIAZ
JAMES M. NEIGHBORS

August 1982

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 Tecnologia (CONACyT), MEXICO.

Table of Contents

List of Figures

1. INTRODUCTION

The goal of this paper is to present a survey of the work related to Module Interconnection Languages (MILs). There exist several languages, software development tools, and operating systems that support some kind of module interconnection. We will focus our attention however, on the languages that are specifically designed to support module interconnection and that are called Module Interconnection Languages.

It is widely known that the technology of software development lags behind the hardware technology and that we are currently in a so called state of "software crisis". The productivity of the software creation process has increased only 3%-8% per year for the last thirty years while the price/performance ratio of computing hardware has been decreasing about 20% per year. Currently the total installed processing capacity is increasing at better than 40% per year [Morrissey 79]. As an example of what this means for just the storage capacities of a medium-priced computing system consider the table in figure 1-1 which extrapolates these trends.

	1972	1982	1992
main memory	32 kilobytes	3.2 megabytes	320 megabytes
secondary memory	5 megabytes	500 megabytes	50,000 megabytes

Figure 1-1: Hardware Capacity Growth over Time

Clearly if these trends continue the organization which purchases a medium-priced computer will no longer be able to develop software specific to the organizations needs which takes full advantage of the hardware. At some point each organization will encounter the programming-in-the-large problem.

Programming a small software system is an essentially distinct and different intellectual activity from that of constructing a large system. In a large system we are mainly concerned with the process of "knitting" system modules together rather than with the process of programming each module. A MIL can be considered a design language because it states how the modules of a specific system fit together to implement the system's function. This is architectural design information. MILs are not concerned with what the system does (specification information), what the major parts of the system are and how they 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 seems to be 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.

Work in this area can be traced back to the early 1960s when the first large software systems like OS/360 started to create real headaches 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.

Modularization was first used as a managerial device to break the work of a big project up into controllable units, and apparently for this purpose the details of the division are not very important. In a software system however, the splitting up is crucial. The connection between modules must be made explicit in order for the system integration to succeed. Early modularization methods focussed on execution-time procedural encapsulation mechanisms such as subroutines and job control languages. Today there are several new development-time techniques that have contributed to better understand and formalize this design process. Structured programming, hierarchical design, functional decomposition, abstract data types and stepwise refinement are some of these newer techniques. These techniques have improved significantly the software development life-cycle and have contributed to the birth of the first MILs.

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 [Estrin 78] 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.

In this survey we will concentrate only on the interconnection problem as seen from the Software Engineering perspective only. 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.

In the first section of this survey the general concepts and ideas that define a Module Interconnection Language are presented. Here we give a definition of what MILs are. The second section presents a brief overview of other systems that support some kind of module interconnection. Many of these systems support more than just the Software Engineering MIL perspective on module interconnection. In the third section four of the MILs developed to date will be described in detail. The MILs are: MIL75 [DeRemer & Kron 76], Thomas' MIL [Thomas 76], Cooprider's MIL [Cooprider 79], and INTERCOL [Tichy 80]. Each one of these MILs is covered in sufficient detail to give the reader a basic understanding of the concepts and ideas behind each one. A description of MESA [Mitchell, et.al. 79], a software development system which supports module interconnection, is included to show its similarities and differences with the MILs. The last section is a general discussion on the future trends of this area of research.

2. MIL GENERAL CONCEPTS AND IDEAS

The fundamental concept of Module Interconnection Languages (MILs) is based on the difference between Programming-in-the-large (PL) and Programming-in-the-small (PS). The primary difference between PL and PS is that "structuring a large collection of modules to form a system (PL) is an essentially different intellectual activity from that of constructing the individual modules (PS)" [DeRemer & Kron 76].

PS is concerned with building programs, with the particular use of loop-constructs, if-statements, assignment-statements, expressions, arrays, and so on. PS 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 is concerned with building systems. We would define a system as being a relatively independent group of programs (modules) which cooperate to implement a complicated function for the organization in which it is embedded. 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.

Using these concepts the specification of a complete system must include three items:

1. A PS (programming language) description of each of the modules in the system.
2. A PL (MIL resource language) description stating the resources provided and required by each module in the system.
3. A PL (MIL interconnection language) description of the resource flow between the modules in the system.

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 primitive operations 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 primitive operation that helps to provide proper module structure within a system as will be shown below. A must attribute may also precede the above operators.

The MIL description of a module specifies the resources required and provided by the module. This module description becomes the interface with other modules and subsystems and is made up of resource names and the operations which act upon them. This is design level information describing system structure and the representations used are quite similar to the design representations used by software engineering methodologies such as Structured Design [Yourdon & Constantine 79]. 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 order to better illustrate these ideas, it could be said that a module is analogous to an **Ada** package with the specification part being the MIL resource description and the implementation part (body) being the code of the module but with the difference (among others) that an Ada body is restricted to the Ada language while the code of modules used in a MIL could be coded in different programming languages or made up of plain text. ADA does not contain a MIL interconnection language for describing a hierarchy of package resource interconnection for a specific system. This function is left to the ADA environment.

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.

An example of a MIL description of a module is shown below. Note that declarations such as **module, function,** and **consist-of** are also part of the MIL syntax. Note that the description code for XA and YBC could be written outside ABC.

```
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
```

Figure 2-1: Example of a MIL Code

2.1 What MILs Do

In this section we will describe the basic functions that a typical MIL should perform. MILs have evolved rapidly in the last few years and newer MILs include other functions which will be regarded here as more advanced. The basic functions are:

1. Describe system structure by defining scope of names across modules and subsystem boundaries and specifying the interconnection between modules. This is accomplished when writing the description part of each module and compiling all the descriptions together.

2. Establish static intra-module connections and do static type checking across module boundaries. Static here refers to compile time while dynamic would mean at execution time. This function is a consequence of the first.

3. Provide for different kinds of accessibility to module resources (e.g. read only, read and write, etc..) and allow modules and/or subsystems to be written in different programming language or to consist of text only.

4. Manage version control and system family. This is an advanced but necessary function in developing large systems.

Note that a MIL needs only the description of the modules, not their body to work, thus effectively separating the activity of PL from the activity of PS.

Aside from these basic operations listed above, 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.

2.2 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. 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.

Some integrated software development systems, such as MESA, PWB, ADAPT, and PROTEL, perform some kind of module interconnection functions (e.g. type checking across modules, etc.) as well as some of the functions stated above as not belonging to the domain of MILs. This is why development systems of this kind are not considered strictly as MILs.

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 GANDALF [Haberman, et.al. 81].

Approaches such as C/MESA of the MESA System [Lauer & Satterthwaite 79] and <u>External Structure</u> 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. The MESA system is discussed in section 4.5.

3. OVERVIEW OF MIL RELATED WORK

3.1 Modularization

Modularity is a well established concept that has been used in engineering disciplines for many years. Modularization has also been used as a managerial devise 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]. Dijkstra introduces ideas like:

- Divide and conquer approach to characterize top-down design by presenting the three main stages of structuring:

    1. Make a complete specification of the individual parts
    2. Verify that the total problem is solved by proper assembly of the above parts
    3. Construct each individual part as specified

- The principle of non-interference which pertains to this "dissection" technique. He points out that:

  > "The correct working of the whole can be established by taking, of the parts, into account their exterior specification only and not the particular of their interior construction."

  Thus

  > "... the individual parts can be conceived and constructed independently from one another."

In [Dijkstra 76] Dijkstra presented the idea of modularization and hierarchy in program design. This idea is presented as the analogy of the pearl string. He suggests that we visualize a program as a string of ordered pearls in which a larger pearl describes the entire program in terms of concepts or capabilities implemented in lower-level pearls.

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. This is what structured design and structured programming is all about. The different possibilities of hierarchy and modularization have been classified as types of coupling and cohesion by Software Engineering [Page-Jones 80].

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 gave birth to the more generalized ideas that triggered the development of Module Interconnection Languages (MILs) of the late seventies.

Although structured programming was characterized by techniques like Stepwise Refinement [Wirth 71] which is also an example of modularization, newer and more advanced techniques like the use of abstract data types in program design [Flon 75], the hierarchical ordering of program segments, and the use of verifiable control structures and operators became more attractive for defining modules.

Parnas has proposed an alternative way of looking at the problem. He uses "information hiding" as the criterion for division into modules. In 1972 [Parnas 72a] he began formalizing modularity by uncovering the whole mystique of modularization, asking questions like: What is a module?, What distinguishes a good module from a bad one?, How do we go about modularizing a program?, etc... He introduces the term information hiding which resembles Dijkstra's principle of non-interference. He illustrates how information hiding is implemented by contrasting two modularization techniques:

1. The conventional way; starting from a flow chart and functional decomposition using formats and table organizations as interfaces.

2. Using information hiding; by defining fairly independent functions that operate on the data.

"Every module in the 2nd case is characterized by its knowledge of a design decision which it hides from all others. Its interface was chosen to reveal as little as possible about its inner workings".

He demonstrates that the second method is less sensitive to changes in requirements and/or design than the conventional method. His fundamental observation here is that a programming task is more manageable if it is partitioned by common decisions, rather than by specific characteristics such as control flow.

In an earlier paper [Parnas 72b], Parnas introduced the idea of viewing programs and designs as a collection of somewhat static and independent objects rather than sequential decision makers. The idea of being able to completely specify a whole system out of a collection of modules is based on this approach.

In [Parnas 76] he goes further by demonstrating that module construction using information hiding is better suited for the development of program families. He compares his technique against the classical way of building modules (sequential completion) and shows that later versions of a program constructed by using the later technique, have performance deficiencies because they were derived by modifying programs designed to function in a different environment. Information hiding used as a development technique is characterized by:

- Precise representation of intermediate stages.
- Postponement of certain decisions while continuing to make progress towards a complete program.

Stepwise refinement is also characterized by these two ideas but uses instead incomplete programs as intermediate representations but with completely specified operators. This technique is better suited for small systems while Parnas' method although harder to implement works better in large systems. Modularity thus has evolved from few sparse ideas to a well defined concept and even to be the base for new software building techniques.

3.2 Modularity by Using Abstract Data Types

Abstract data types have been used to represent modules. Liskov and Zilles ([Liskov & Zilles 74], and [Liskov & Zilles 75]) being one of the earlier advocates of this idea, proposed the use of abstract data types together with the discipline of structured programming as a feasible technique to build correct programs.

Here a programmer is concerned with proving that his program is correct by writing a program which solves the problem but which runs in an abstract machine. This top level abstract machine must provide just those data objects and operations which ideally solve the problem. Most of these data objects and operations are truly abstract and are not present as primitives in the programming language being used.

The programmer analyzes the way his program makes use of the abstractions but not with any details of how those abstractions may be realized. Once satisfied with the correctness of the program, the programmer turns his attention to the abstraction it uses. Each abstraction represent a new problem, requiring additional programs for its solution. The new program may also be written to run on an abstract machine, introducing further abstractions. The original problem is completely solved when all subsequent abstractions are realized by programs.

A programming language (**CLU**) was later designed [Liskov et.al. 77] to implement this methodology by providing 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. For example, what kinds of abstractions can be defined by each. For brevity only CLU will be described here.

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 <u>description unit</u> which contains all system-maintained information about that abstraction. The <u>interface specification</u> 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 the first MIL [DeRemer & Kron 76], MIL75. In a MIL, as will be shown later, each module has a specific separate description of the resources required and produced (Module Description) which becomes the interface with other modules and subsystems.

The best 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 of system organization. It will be shown latter that a MIL based on a compiler is not as effective in the control of a system organization as a MIL based on a data base processor [Cooprider 79]. It could be argued 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) [DeRemer & Kron 76] at a level lower but similar to MESA [Geschke et.al. 77]than a LPL (Language for Programming in the Large).

3.3 Modularity by Using Nonprocedural Descriptions

Nonprocedural programming languages are used mainly as very high level languages (VHLL) for system specification or in the first program description in automatic program generation. From the viewpoint of MILs, there is however, a nonprocedural programming language (**MODEL**) that supports module development.

The MOdule DEscription Language (**MODEL**), was created by N.S. Prywes [Prywes 77a], [Prywes 77b], and [Prywes et.al. 79] as one of the bottom up building blocks towards the full automatic generation of programs.

MODEL is a complete system that helps the user by interactive dialogue to build **processable** modules. It is non-procedural and uses a data base that verifies completeness and consistency of the module requirements.

Each module is composed (in any order) of:

- -Header (module name, source and target data names and, references).
- -Data Description (file description and inter file pointers).
- -Computation Description (interim parameter description, source set assertions and, target set assertions).

To avoid sequential ordering of data and computations it uses assertions in data descriptions and in computations. Statically then, completeness is checked and code for each module can be generated.

The objective here for creating modules, is to be able to generate code for each module independently. The user enters the specifications of the system to be constructed in MODEL statements. Syntax is analyzed and then the non-procedural specification statements are made sequential using a complicated algorithm based on a precedence matrix and on graph analysis. The sequential high-level code is then checked for completeness and fed back to the user for modification of requirements. This cycle is repeated until the user is satisfied. Finally the high-level sequential code is converted to programming code.

MODEL´s contribution to MILs is the concept of non-procedural modules that provide less coupling among modules and increases the capacity to have **reusable** modules. MODEL is excellent for describing modules but it has no provision for module interconnection.

Sangal, in his Ph.D. thesis [Sangal 80], introduces a new language (NOPAL) intended to provide MODEL with a module interconnection capability. Sangal uses the same approach (as Prywes) to modularity but he introduces the idea of data abstraction to specify abstract data types and the functions that are allowed to operate on variables of the data type.

In NOPAL, each of the modules can be specified and processed by the language processor independently. Communication between modules is by means of abstract data types. A module represents an abstract data type that can be used by other modules, thus allowing recursive usage.

NOPAL may be considered as a cross between MODEL and CLU but with even less capacity of system organization than CLU. Control of module interconnection is implicit, not explicit as in MILs. NOPAL is also a LPS and cannot handle different versions or system families. It must be recalled that NOPAL is a tool designed for automatic program generation just like MODEL, and its capabilities as a MIL are limited.

3.4 Tools Supporting Module Interconnection

There exists several tools that in one way or another support some sort of module interconnection mechanisms. In this report, only some of these tools which are closer to MILs, and widely used will be briefly described.

To describe every tool, system or methodology that supports some kind of module interconnection is beyond the scope of this paper. Systems like IBM´s JCL and many interactive command languages that process compile, link, load and execute commands perform to some degree some kind of module interconnection but, they do not fit all the requirements for MILs as presented in section two.

3.4.1 Job Control Languages

The use of JCL to bind a system together is a pitfall of system design. With this scheme the system is written as a group of independent programs which exchange information through assignments in the job control language. A disk file is the easiest method for passing information in most JCLs. There are two major problems with this scheme.

First, the system is "linked" together each time it is run. The inefficiency of this action is not the problem but the fact that the integrity of the system relies upon the JCL linkage is a problem. Evolution or maintenance on the JCL processor may cause old systems not to execute properly because the details of the system interconnection are not checked. Maintainers of control languages know this and thus have a tendency to add features rather than change the command language. This causes the command language to become very complex.

Secondly, the execution time linking of these systems gives rise to a dangerous style of system evolution, that of building "layers" around an existing complex system. The layers are preprocessing and postprocessing passes over the input and output data of a system to modify the systems behavior without knowing how the system specifically uses the data. These layers are accompanied by some modification to the constituent programs of the system. These "filtering" passes are easy to add to a JCL bound system and very hard to remove. Thus, the effective size of the JCL control increases and the number of programs involved in the system increases. This increase in the size of the system aggravates the problem even more. Filtering passes added to the interior of a system are even harder to deal with.

3.4.2 UNIX

UNIX is by far the most widely used programming environment that can be classified as a group of programming development tools [Ritchie 79], [Ritchie & Thomson 74], and [Kernigham & Mashey 81]. The UNIX development environment is essentially a timeshared operating system with an extensive set of tools. All of the tools and the underlying operating system were built with a highly consistent design philosophy that uses a single uniform file format almost exclusively. This facilitates the use of various tools in conjunction with one another.

UNIX does not have any **explicit** commands that support module or program interconnection, nor any support for type-checking across modules. A designer can build a complete system out of UNIX commands alone but, program correctness, parameter passing consistency, and system completeness must be manually verified. UNIX could be seen as a very sophisticated and interactive JCL which by its own nature is inappropriate for MIL tasks.

3.4.3 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. 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 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 is the version control mechanism that inspired some of the system version control techniques used by the new MILs.

3.4.4 CLU

CLU [Liskov et.al. 77], although being a language by itself, their support facilities fall into the category of tools. From this point of view, CLU provides a better structure for module interconnection than UNIX and PWB. The CLU system has a central data base of system information, including the modules (realizations) and the associated descriptive units and module interfaces. The entire interconnection scheme of a system can be determined by tracing the system from the root description unit, determining the set of modules used by it, and proceeding recursively on each of those modules.

The CLU mechanism permits the definition of multiple versions of modules in the sense that module definitions can be copied and the associated descriptive unit modified to include a different resolution of the modules used by that module. This approach however, is not practical because of the amount of extra storage required to save redundant information.

A deficiency of CLU is the restriction of the interconnection mechanism to CLU -style modules which, as mentioned above, permanently binds a module to the abstraction it uses. This limits its version control capabilities. In CLU the overall interconnection structure of who knows whom within a collection of modules is deducible only by examination of the compiler inputs for all modules.

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 and has been reported to be an excellent tool to control overall system design and decomposition. It is used to control inter-module type-checking during compilation. It is an automated resource, interacting with the ADAPT compiler.

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. A more detailed description of this approach is presented in section 4.5.

3.4.5 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].

Besides having the same disadvantages of MESA, (see section 4.5.3) PROTEL is very limited in controlling system versions and in supporting system organization.

3.4.6 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 **MISC** (Module Interconnection Specification Capability) is intended to enhance the power of SARA by providing a smoother path from modeling to code.

In this new model a SARA-MISC mapping is obtained in which the SL1-GMB model identifies the variables and the calls of the code; the MIL model identifies the type and procedure definitions; and the mapping (SARA-MISC) says which variable is of what type and which call is of what procedure.

As of 1979 the SARA-MISC 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.

3.4.7 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 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 <u>syntax directed editor</u> and a <u>syntax directed dynamic debugger</u>. 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 <u>incrementally</u> 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 [Cooprider 79] and later by Tichy [Tichy 80]. 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 72a] 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 in contrast with other new software development environments, has been implemented and is currently under evaluation.

Figure 3-1 below illustrates graphically the relationship among the tools described above and their support of some kind of module interconnection.

3.5 Other Ideas about Module Interconnection

Parnas in [Parnas 78] and [Parnas 79] integrates most of his ideas (information hiding in particular) about modular construction of programs and proposes a new methodology for better system structure.

This new approach is based on the <u>virtual machine</u> concept by which a system designer must stop thinking of systems in terms of components that correspond to steps in the processing. He must find instead, the primitive operations of the system (specific domain analysis [Neighbors 80]) and build one machine on top of another. At each step take advantage of the newly defined "primitive operations" and this step by step approach turns a large problem into a set of smaller ones. A positive side effect of this methodology is that each element in

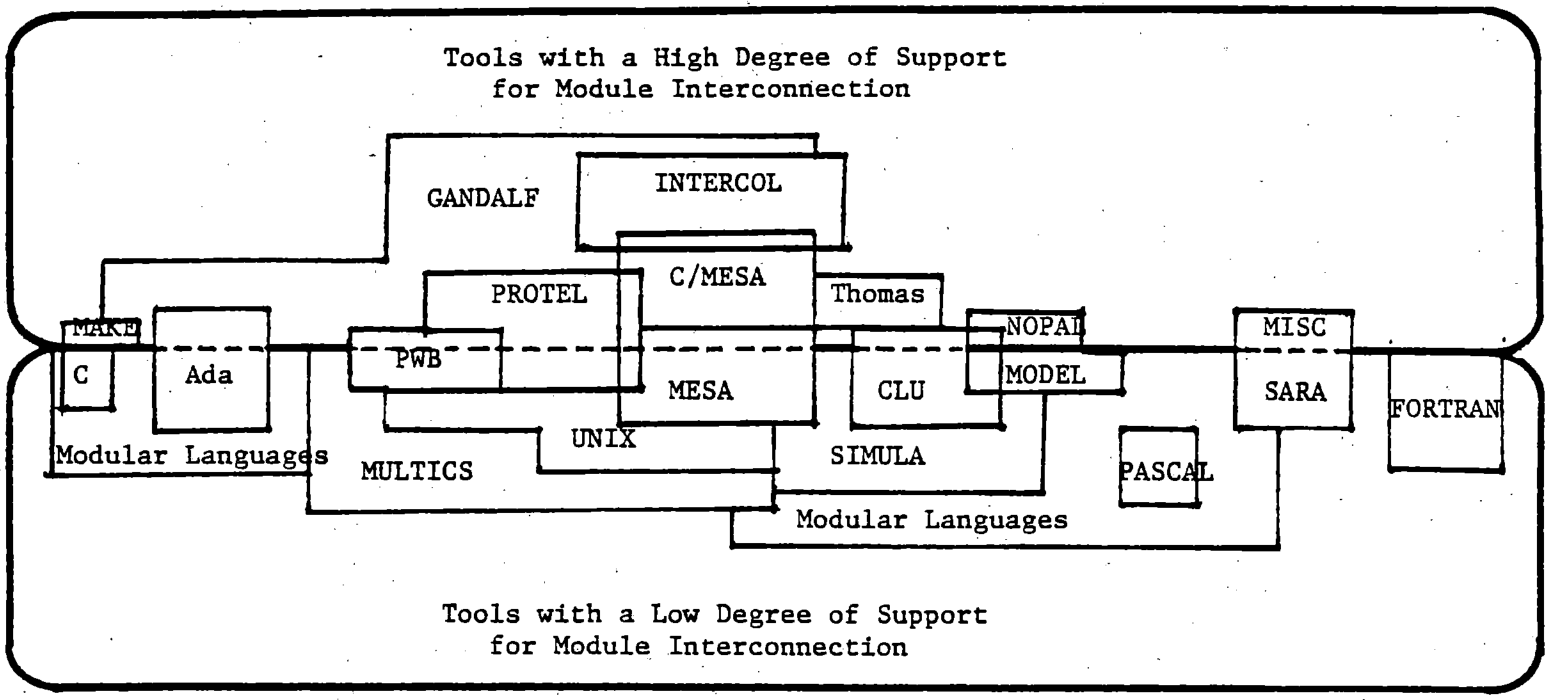

Figure 3-1: Some Tools Supporting Module Interconnection

this set of virtual machines is a useful subset of the system and can be reused somewhere else.

In order for a higher level virtual machine to execute properly, its lower level virtual machines must have been executed first and so on until reaching the first level hierarchy. To establish this proper order of execution priorities, Parnas introduces the uses structure which establishes the relations among programs using others. This implies that different uses structures can lead to different systems while using some of the same primitive modules.

This approach could be considered as a MIL in which the virtual machines at different levels are the modules and the uses structure is the interconnection specification (interface) among them. This approach however, could be called vertical interconnection and not horizontal as in the typical MIL, thus leading to interesting questions about its performance and possibility of being automated as a regular MIL. Parnas claims that because of the insight required in constructing virtual machines on top of others, automatization of this building process would be too restrictive.

A more radical approach to module interconnection is advocated in [Belady & Lehman 79]. Belady agrees with the idea that "the interface between software units, at least in any one environment, be standardized and implemented in hardware." This idea is based in the "Funnel" concept [Lehman 76] and supported by the advancing mini-computer and micro-processor technologies which tend to overcome the limitations of standard hardware interfaces.

This approach seems to impose an unnecessary constraint on inter-software unit communication by seriously deteriorating system performance and although Funnels (Functional Data Channels) can be implemented in microprocessors, it is a difficult problem to carry such concepts into a software module. The feasibility of this approach thus, relies on the proper solution to the specification and interface problem.

4. DESCRIPTION OF THE EXISTING MILs

There are several languages and software development tools that support some kind of module interconnection as mentioned in section 2 and 3 above. Some of the software development tools like PROTEL, could be considered almost as having a MIL except for failing to meet one or two of the characteristics defined in section 2 as basic functions of a MIL.

In this section the four stand-alone MILs developed to date and reported in the literature will be presented: MIL75 [DeRemer & Kron 76], Thomas' MIL [Thomas 76], Cooprider's MIL [Cooprider 79], and INTERCOL [Tichy 79] together with MESA, a software development system that supports module interconnection in a fashion similar to a MIL. For each one of these MILs, a brief introduction is presented followed by the main objectives of the language and the basic concepts upon which it was built. A short section which presents the main differences of the described MIL from the others is also included together with a note on the experience to date. In Appendix A an example is given to illustrate how a group of modules can be represented in some of the MILs described.

It will be observed below that MILs are one of the products of the "Structured Programming" school but for programming in the large. Also it will be observed that each MIL is built on most of the concepts of the previous one and only some features are added to each. The exception being MESA with similar features but implemented differently. This idea is illustrated below in figure 4-1.

4.1 DEREMER AND KRON'S MIL75

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.

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. The designer must specify for each module, the resources provided and required as well as their type; the details about their internal operations are not required. MIL75 compiles all these specifications while doing consistency checking resulting in an accurate recording of the overall solution structure.

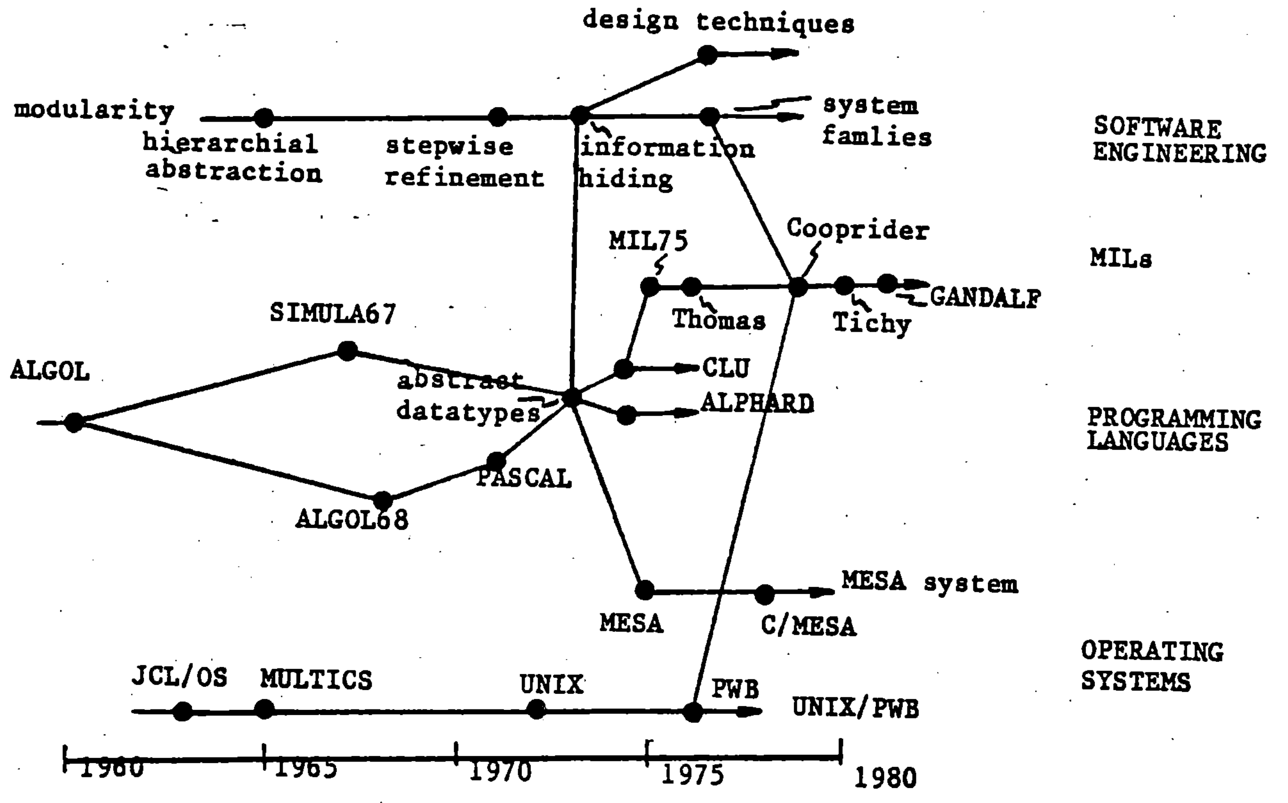

Figure 4-1: Graphic View of MIL Evolution

4.1.1 Objectives

The main objective of MIL75 was to provide some means of programming-in-the-large by describing interconnection among modules. It was intended to serve as project management tool and as design tool by helping to provide a means to present and verify whole system structure before starting construction. As a side effect, having a standard, formal, concise and verifiable means of representing system structure, MIL75 would provide a natural means of communication between members of a programming project and means of documenting system structure.

4.1.2 Basic Concepts

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 4-2 below shows a system tree for a one-pass compiler.

   The system tree is a rough hierarchical decomposition of the intended software project where each node (module) should encompass an intellectually manageable part of the whole problem and may include more than one program or function. The scope of definitions thus aids the systems designer in determining the hierarchical levels of the system tree.

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 later 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 as will be shown below.

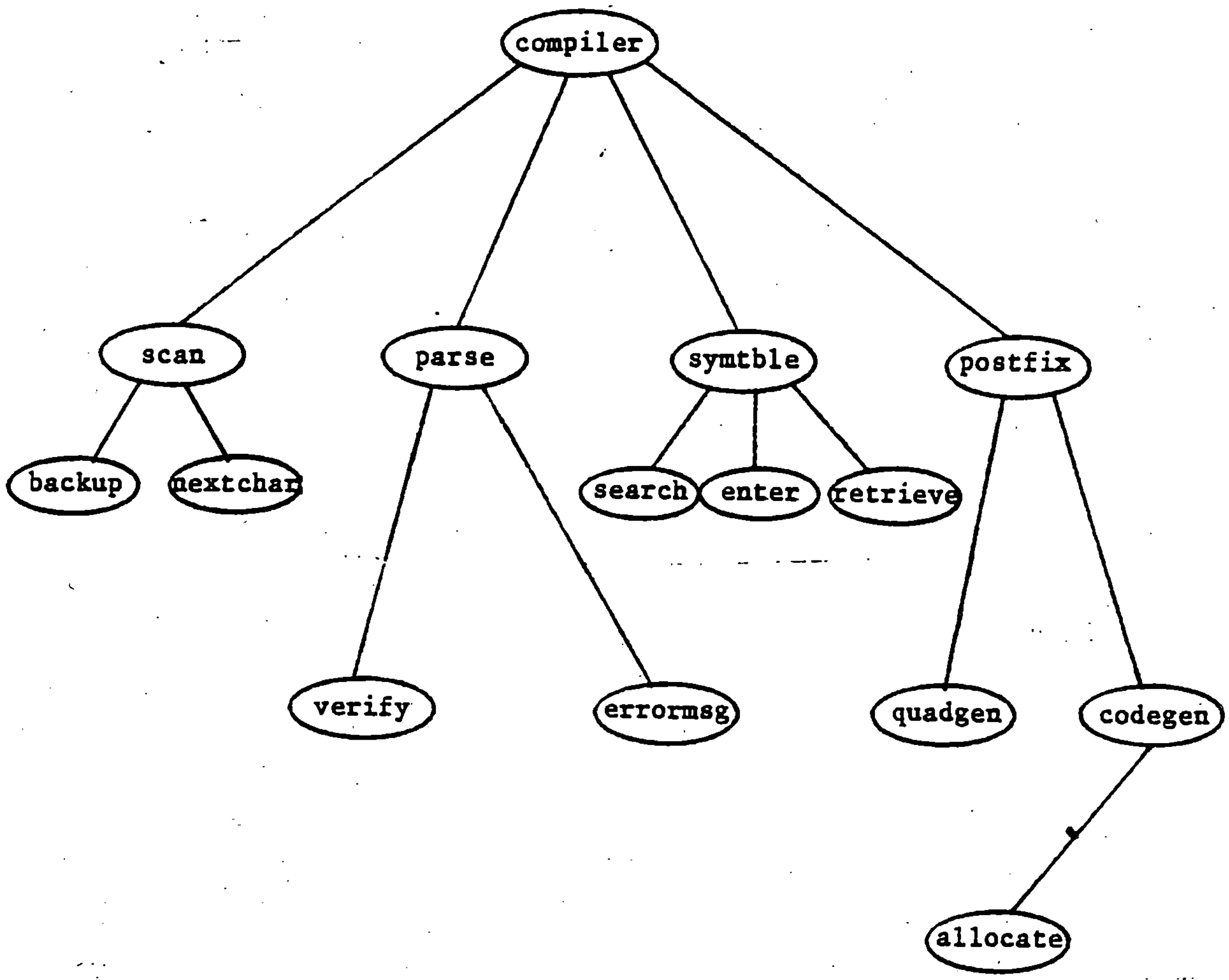

Figure 4-2: Graphical System Tree for a One-pass Compiler

The resources provided by a node (module) must come from its children, that is, a node can only provide resources to its parent if they have been either derived by one of its children or defined within that node. A module cannot provide resources that have been acquired from other nodes outside the direct family genealogy. The systems designer is responsible for including such outside resources within that node.

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 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.

   Accessibility in MIL75 is defined as: "A node p has access to a node n iff either p has sibling-access to n, p inherits access to n, or p is the parent of n" [DeRemer & Kron 76]. So access to a resource by a given module is either unrestricted or none at all.

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 4-3 below.

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 4-3. For this example the access augmented system tree mentioned above is identical to the module interconnection structure of Fig. 4-3 except for the usage links.

In short, a complete MIL75 program consists of a series of statements expressing the different relationships (1,...,5 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 above is given in Fig. 4-4 to illustrate the above discussion.

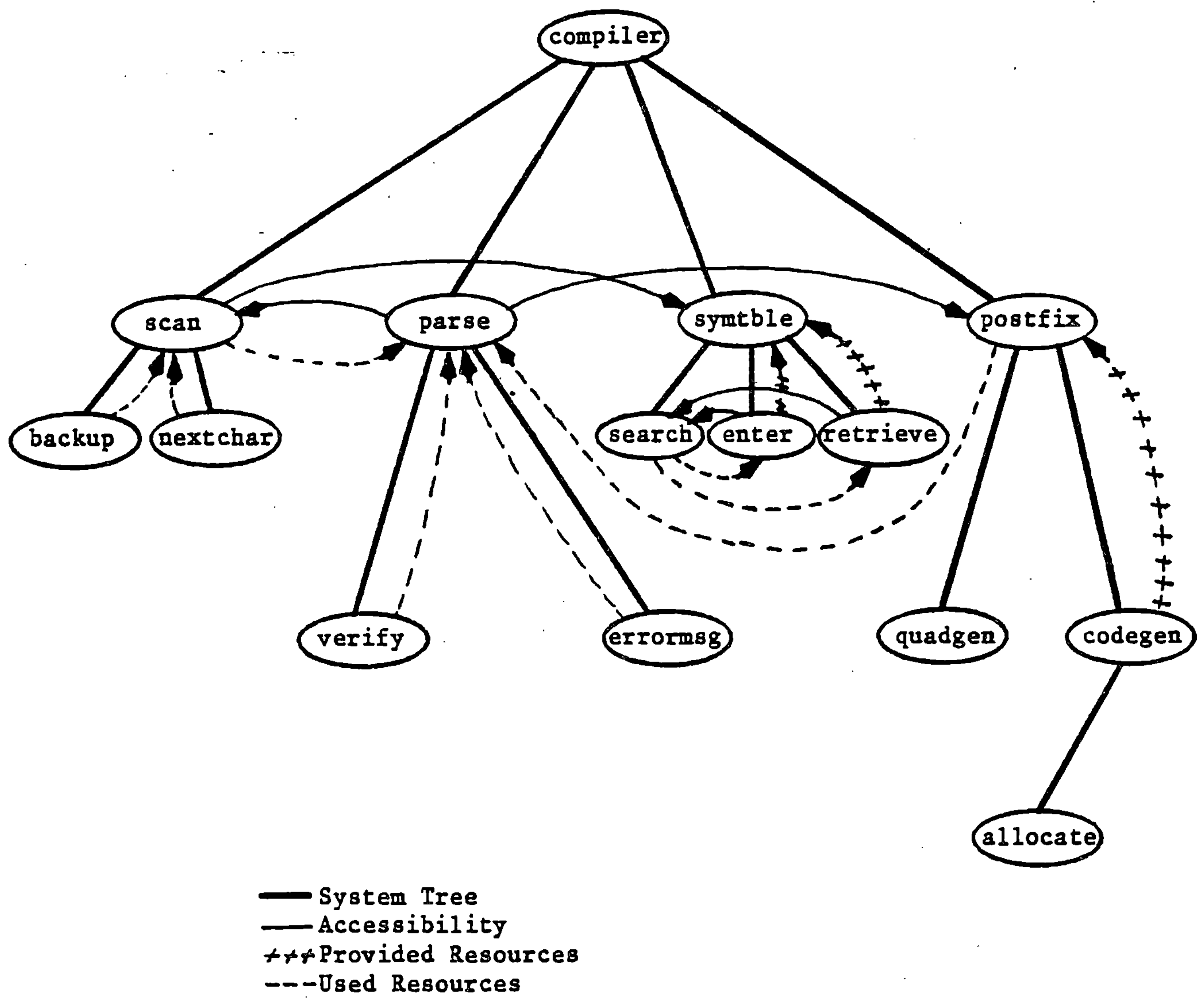

Figure 4-3: The Module Interconnection Structure

```
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 4-4: Partial Code of a MIL75 Program

4.1.3 Differences from the Other MILs

Before an attempt is made to point out the differences between MIL75 and the other MILs we must recall that MIL75 was the first MIL and that most of its characteristics were original and innovative at its time. Later MILs improved MIL75 ideas, added new ones, and indicated some of the disadvantages of MIL75.

The highlights of MIL75 are:

1. First case of a compiler used for a LPL.
2. Parents control sibling access thus providing an effective tool to do design based on virtual machines. Hierarchical levels on the structured model are effectively separated.
3. The scope of control in the design is facilitated by parents making resources available (by default) to offsprings and side effects are minimized by not allowing resources to bubble-up unless specified

MIL75 is oriented around a structured (oriented tree) representation of a system thus shifting somehow 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.

4.1.4 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.

4.2 THOMAS' MIL

Thomas developed a module interconnection notation and discussed a possible module interconnection processor [Thomas 76]. He proposes a formal model based on the compile-->bind/link paradigm 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.

4.2.1 Objectives

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. His MIL is designed to meet many of the characteristics of CLU and ALPHARD. Thomas' MIL is intended to be used for specifying structural models of software systems as well as for documenting module interconnection structures.

Thomas also proposes a flexible binding scheme for the interconnection of the modules and the incorporation of the MIL into a programming system or environment. The objectives of these propositions are:

1. The ability to express the mutual support of modules explicitly by

combining sets of modules together in a fashion similar to how CLU and ALPHARD use their language extensions to support higher level language constructs.

2. To allow the systems designer certain control over the distribution of information.

3. To provide the capacity for arbitrary graph structures of accessibility relationships and at the same time control the spread of this accessibility.

4. To realistically support modularity and facilitate the <u>reuse</u> of already developed subsystems through the support of a structured library of predefined modules.

As has been shown above, objectives 1 and 2 above were partially fulfilled by MIL75 while 3 and 4 are the innovations contributed to MILs by Thomas.

4.2.2 Basic Concepts

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" schema as illustrated below:

```
checking
   |
   |         |
   `--------->|
             |
   ,--------->|
   |         |
   |       binding/linking
compiling
```

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 as he claims, a software system to be represented (in MIL code) as a "finite directed graph G with no simple cycles and where S is a <u>start node</u> 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.

From the MIL75 point of view, this binding operation is what the MIL75 compiler does when transforming the "access augmented system tree" to the "module interconnection structure". The advantage in Thomas' MIL is however, to allow more freedom to the systems designer to concentrate in doing good modularization in the terms proposed by Parnas [Parnas 72a] (i.e. information hiding) and delegate some of the structuring to the MIL processor. This approach also allows as mentioned before, the effective use of a structured library for module reusability. A drawback however, is that its flexibility is still very limited.

A MIL from Thomas viewpoint should be called an "Environment Establishment Language" because it specifies the bindings of names to modules which provide those names (resources). According to him, an environment is a set of names accessible for use in a program where this set of names characterize a virtual machine. In MIL75 terms this "environment" is called the "module description".

He advocates a static binding scheme similar to ALGOL and other structured LPSs but avoiding some of their deficiencies like: automatic inheritance, implicit and fixed accessibility, forcing a variable (resource) upwards to a common area to increment its accessibility, etc. An argument he uses against dynamic binding besides cost is the advantage of having static type-checking.

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

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

- <u>Modules</u> are units of source code (may be written in different programming languages) providing and requiring resources. This MIL allows for two kind of modules: <u>procedures</u> and <u>clusters</u>. Procedure modules provide a single name used to reference the module as a whole (e.g. function call). Cluster modules provide both a name for the whole module and (optionally) a set of entry points (e.g. data abstraction like in stack$pop).

  The definitions of resources and modules are almost identical to the ones given in MIL75.

- <u>Nodes</u> 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.

In MIL75 each module has a "node" or each node belongs to a module, in Thomas´ MIL a node may encompass or "describe" more than one module.

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. Only this set of resources should be known to the user and nothing of the internal structure of the subsystem. A subsystem is stored in a library structure and can be referenced in a MIL program as if it was a single node.

  This concept of subsystem is to make explicit what MIL75 implicitly uses for system design based on layers of virtual machines.

The concrete syntax for this MIL is presented in Appendix B. The figure below shows a piece of code for this MIL to illustrate the use of the names defined above and how they describe a structure. In Appendix A a complete example is presented.

If a user were to design a software system using Thomas´ MIL as a development tool, he/she should follow a structured design methodology to obtain an oriented tree structure of the system just like the "system tree" of MIL75 is obtained. Then he/she would 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

```
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 4-5: Example of Code for Thomas′ MIL

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.

4.2.3 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.

4.2.4 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.

4.3 COOPRIDER'S MIL

Cooprider expands the basic ideas of the previous MILs to introduce a version control facility and a software construction facility [Cooprider 79]. The former facility recognizes the different instantiations (versions) of an interconnection network and knows how they are hierarchically integrated while the later 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.

4.3.1 Objectives

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 [DeRemer & Kron 76] and Thomas [Thomas 76].

From the MILs point of view, Cooprider's work could be considered to be a MIL extended to support version control and system construction by adding new primitive operations like, among others, <u>version</u> and <u>acquire</u>. By extending the domain of operations of this MIL, Cooprider presents convincing arguments that justify the use of a central data base processor against the use of a compiler as previously suggested in the earlier MILs.

The goal of Cooprider's extended MIL is to effectively represent:

- Subsystems that provide and require resources.
- Versions of subsystems that share the interconnection structure.
- The resources that may be needed for the construction of any subsystem version.
- The construction rules that operate on the concrete system components.
- The execution of a construction program.

4.3.2 Basic Concepts

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. Appendix B (section II.2) shows the interconnection syntax of 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 KWIC [Parnas 72a] example of Appendix A (section I.2) illustrates these three mechanisms and the use of the interconnection syntax.

-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 syntax of the construction process is presented in Appendix B

Class No. ARCHIVES	AUTHOR Prieto-Diaz, Ruben
List Price Z	TITLE MODULE INTERCONNECTION LANGUAGES: A SURVEY.
Date Ordered 699	
Date Rec'd. C3 9-30-85JM	Edition or Series (Calif. Univ. Irvine. Dept. of Information & Computer Sci. Tech. report ; 189) Volumes
Dealer NO. 189	Place Publisher Year Irvine : Dept. of Info. & Computer Sci., Univ. of Calif., 1982.
C. 2	Recommended by Fund Charged Cost
No. of Copies 2	Bk. Hr. from Serials
Order No.	HANDLING : 1 copy-stax 0-cui 1 copy- archives 0-pf

(section II.2). 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 is illustrated in Appendix B (section II.2). It 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, he/she would follow a similar process as if using MIL75 or Thomas´ MIL, that is, follow a structured design methodology, specify each module and their hierarchical interconnections and represent this module (subsystem) structure in MIL syntax. 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 be not 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.

4.3.3 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.)

4.3.4 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].

4.4 TICHY′S INTERCOL

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.

4.4.1 Objectives

The goal of Tichy′s work is to provide a programming environment that insures the consistency of a programmed system at the module interconnection level. Although this goal is what has driven the previous work in this area (as seen above), he succeeds in extending, improving and integrating the original ideas. His focus is mainly on the Ada language. At the software development environment level he 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. This facility should supervise interface changes by updating the affected components, alerting the programmers, and preventing the use of inconsistent 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. This subsystem should also be capable of handling compilation, recompilation, and integration.

The INTERCOL processor and the two systems described should be integrated into a single environment.

At the MIL level, he aims at several innovations of the notation used by the earlier MILs. With the new features he expects to expand the language so that the interface and version control system instructions can be programmed using the same notation. The new or extended notation pertains to: multiple versions, multiple configurations, multiple programming languages, and support for software development control.

4.4.2 Basic Concepts

A description in INTERCOL is a sequence of module and system families followed by a set of compositions. A member of a <u>module family</u> is a version of a module, and a member of a <u>system family</u> is a version of a system. The former may be one of a set of different module <u>implementations</u> for different environments or in different languages, or may be one of a set of different module <u>revisions</u>, or can also be a <u>derived version</u>. The later may be a member of a set of different system <u>configurations</u> or of a different <u>derived composition</u>.

Each one of the above families has an <u>interface</u>. An interface consists of programmed entities called <u>resources</u>. 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 <u>resource-specification sublanguage</u>. 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 specific list of resources written in the resource specification sublanguage. The syntax of a resource declaration (see Appendix B, section II.3) is more powerful and allows for more degree of preciseness than the syntax used for resources in the previous MILs but at the same time demands more attention of the designer in order to make a correct specification.

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. The syntax for module families can also be seen in Appendix B (section II.3). 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 subimplementation is intended to give a name to a subset of an existing realization by fixing the revision with an extension. This reduces the cardinality of the family.

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. See Appendix B (section II.3) for the system family syntax.

To show how the concepts just described combine into an INTERCOL program, an example is provided in Appendix A (section I.3) where Parnas' KWIC Index System described in INTERCOL is presented.

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.

4.4.3 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.

4.4.4 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, all that has been proved is that it is feasible.

4.5 XEROX MESA

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. It 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.

MESA was developed at XEROX 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].

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 and is used in the MESA system to specify how separately compiled modules are to be bound together to form configurations.

4.5.1 Objectives

The designers of MESA were interested in creating a language to be used for the production of real system software "right now". MESA thus was not intended to be a MIL but a programming language capable of supporting program modularity in ways that permitted subsystems to be developed separately and bound together with complete type safety.

The language it uses is similar to Pascal or Algol 68 and with a global structure similar to that of Simula. It has its own syntax and some new ideas. 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.

From the MILs point of view, MESA and C/MESA form a well integrated set of tools analogous to GANDALF but covering only the design and implementation aspect of the complete life-cycle of a software system. The MESA System, although intended to be used in a production environment, 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 (the MIL extension of 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.

4.5.2 Basic Concepts

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. Abstractions are implemented by some of the program modules called implementors.

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 and can be partitioned into two parts: a static part and a dynamic part. The first declares types and constants to be shared between client and implementor and the second defines the operations available to clients importing the interface. Only the names and types of operations are specified in the interface, not their implementations. Figure 4-6 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. No knowledge about any implementors of the interfaces is required, but the compiler checks the types and parameters of all references to an interface. The compiler ensures that the types in the exporter (implementor) are completely compatible with the types expected by the importer of the interface.

```
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>;
  pl:PUBLIC PROCEDURE[i:INTEGER] = <code for pl>;
  ....
  pi:PUBLIC PROCEDURE[x:it] RETURNS[y:rt] = <code for pi>;
  ....
  END
```

Figure 4-6: 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 4-7 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 (--).

4.5.3 Differences from the Other MILs

As it can be observed by the reader, 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 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 4.6.

```
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 4-7: A Partial Configuration Description in C/MESA Taken from [Geschke et.al. 77]

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.

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.

Another difference is that MESA modules for example, are restricted to the MESA language thus inhibiting the use of modules written in different programming languages. In MESA a small change to one or few interfaces may trigger a recompilation of an entire system and the configuration description language does not allow versions of systems that vary in their specification nor provides a design level description of interconnections. These disadvantages are the result of MESA not intended to be a MIL but rather a program development tool capable of aiding the development process of medium size projects.

4.5.4 Experience to Date

Experience on the use of MESA has been reported in [Geschke et.al. 77], [Lauer & Satterthwaite 79], and [Horsley & Lynch 79]. The MESA System has been used successfully in the development of an operating system amounting to about 25 thousand lines of MESA code. Several hundred thousand lines of stable MESA code had been written by the end of 1978 by just a group of users at XEROX. C/MESA has also been used as a specification language prior to system implementation.

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. A generalization of the construction process can be represented by a MIL and organized around a unified data base.

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. Such capability has been successfully implemented in GANDALF.

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.

As the reader may be able to observe, 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 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 77b] 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 and for his valuable comments while reading the first drafts of this work. We also wish to thank Ruben´s wife Haydée for her editorial assistance.

I. APPENDIX A -- MIL Examples: Parnas' KWIC Index System

In order to better illustrate the use of the MILs described in this report, the KWIC Index Production System as described in [Parnas 72a] is presented in three different MILs. The first section of this appendix shows the KWIC example as represented in Thomas' MIL. This particular example was taken from Thomas' Ph.D. Thesis [Thomas 76]. The example from the second section was taken from Cooprider's Ph.D. Thesis [Cooprider 79] and the last illustration from Tichy's Ph.D. Thesis [Tichy 80].

The KWIC (Key Word In Context) index system accepts an ordered set of lines, each line is an ordered set of words and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order. The figure below shows the general structure of the KWIC system as proposed by Parnas [Parnas 72a].

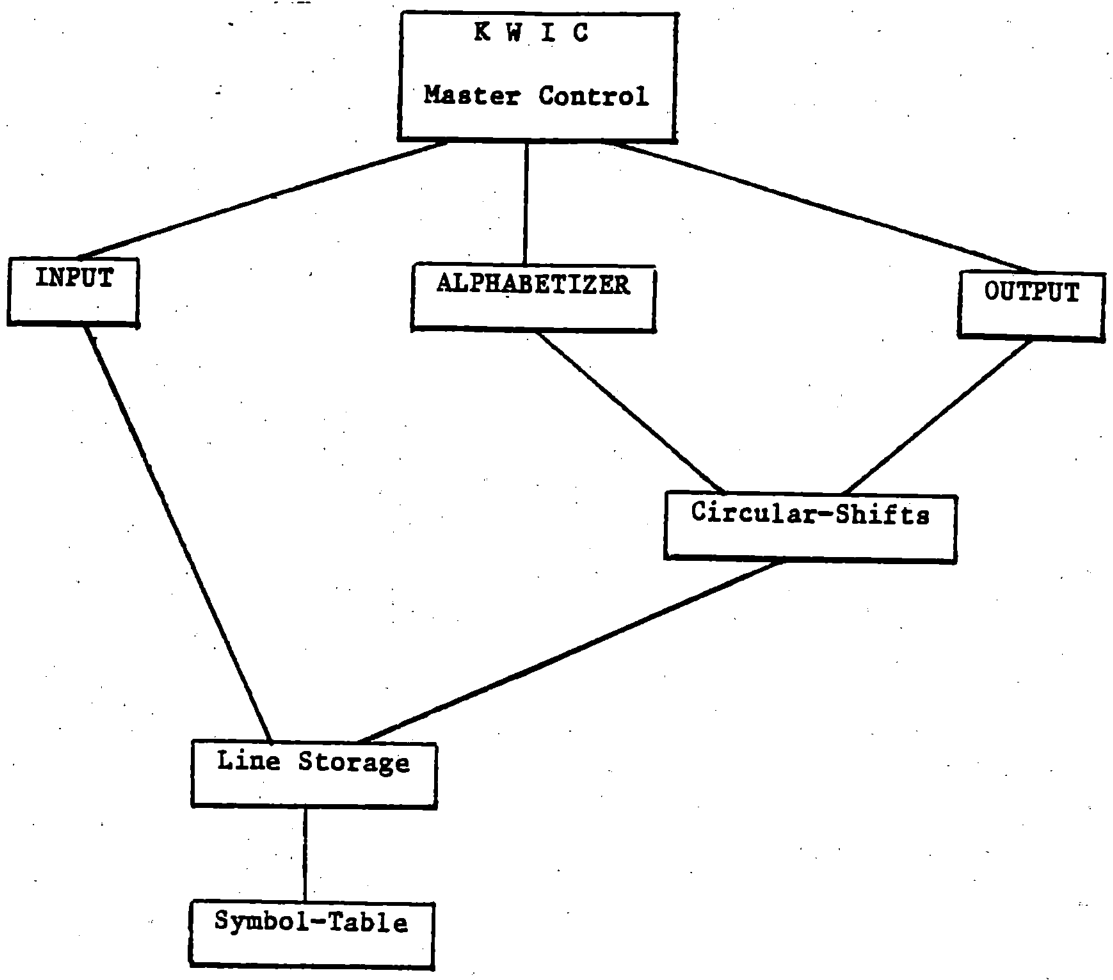

Figure 6-1: General Structure of the KWIC Index System

I.1 KWIC Description in Thomas' MIL

```
node Main
  synthesizes proc KWIC
  generates locally proc KWIC using MC
                    proc input_error_handler using IEH
  has successors Input, Alphabetizer, Output

    successor Input
      must synthesize proc input_lines
      inherits proc input_error_handler

    successor Alphabetizer
      must synthesize proc alph
                      proc ith

   successor Output
     must synthesize proc output_lines
     inherits proc ith

node Input
  synthesizes proc input_lines
  must inherit proc input_error_handler
  generates locally proc input_lines using IL
                    proc storage_error_handler using SEH1
  has successors Line_storage

    successor Line_storage
      must synthesize cluster line with ops setchar
      inherits proc storage_error_handler

node Alphabetizer
  synthesizes proc alph
              proc ith
  generates locally proc alph using ALPH
                    proc ith using ITH
  has successors Circular-shifts

    successor Circular_shifts
      must synthesize proc cssetup
      cluster shifted_lines with ops cschar, words

node Output
  synthesizes proc output_lines
  must inherit proc ith
  generates locally proc output_lines using OL
  has successors Circular_shifts

    successor Circular_shifts
      must synthesizes cluster shifted_lines
                                 with ops cschar, words
```

```
node Circular shifts
  synthesizes cluster shifted_lines with ops cschar, words
              proc cssetup
  generates locally cluster shifted_lines using SL
                  proc cssetup using SETUP
                  proc storage_error_handler using SEH2
  has successors line_storage

    successor line_storage
      must synthesize cluster line with ops char, words, setchar
      inherits proc storage_error_handler

node line_storage
  synthesizes cluster line with ops char, words, setchar,
                                  deline, delwrd
  must inherit proc storage_error_handler
  generates locally cluster line using LS
  has successors Symbol_table

    successor Symbol_table
      must synthesize cluster sym_tab

node Symbol_table
  synthesizes cluster sym_tab
  generates locally cluster sym_tab using ST
```

I.2 KWIC Description in Cooprider's MIL

```
subsystem KWIC provides Kwic requires InputLine, Alph, Ith,
                                        OutputLine
   subsystem INPUT provides InputLine
        requires Line, InputErrorHandler external LS, IEH
        realization...end INPUT

   subsystem LS provides Line
        requires StorageErrorHandler external SEH
        realization...end LS

   subsystem ALPH provides Alph, Ith
        requires CsSetup, ShiftedLines externalCS
        realization...end ALPH

   subsystem CS provides CsSetup, ShiftedLines
        requires Line external LS
        realization...end CS

   subsystem OUTPUT provides OutputLine
        requires ShiftedLines, Ith external Alph, CS
```

```
          realization...end OUTPUT

     subsystem IEH provides InputErrorHandler
                  realization...end IEH
     subsystem SEH provides StorageErrorHandler
                  realization...end SEH
     realization...end KWIC
```

I.3 KWIC Description in INTERCOL

```
----------------------------------------------------------------------
--KWIC accepts a set of input lines (titles) and outputs a listing
--of all circular shifts of all input lines in alphabetical order
--(A KWIC index)
----------------------------------------------------------------------

module LINE-STORAGE
   --LINE-STORAGE manages the storage of lines (titles).
   --It hides the storage and retrieval technique.
   provide package Line is
        constant   MaxLine, MaxWord, MaxChar:INTEGER;
        type       LineN is range 1..MaxLine;
        type       WordN is range 1..MaxWord;
        type       CharN is range 1..MaxChar;
        readonly   Lines:LineN;
        function   Words (Ln:LineN) return LineN;
        procedure  PutChar (Ln:LineN, Wn:WordN, Cn:CharN, C:CHARACTER);
        function   GetChar (Ln:LineN, Wn:WordN, Cn:CharN) return
                      CHARACTER;
        procedure  DelLine (Ln:LineN);
        procedure  DelWord (Ln:LineN, Wn:WordN);
   end Line;
   require exception Storage-Error-Handler;

   implementation
        Core.ada      --stores lines in core
        SAM.for       --stores lines in sequential files
        ISAM.for      --stores lines in index sequential files
end LINE STORAGE

----------------------------------------------------------------------

module CIRCULAR-SHIFTS
  --CIRCULAR-SHIFTS generates the circular shifts of the lines
   require Line.{MaxLine, MaxWord, MaxChar, LineN, WordN, CharN,
   Lines, Words, GetChar}

   provide package Shifted-Lines is
        type       ShiftN is range 1..MaxLine*MaxWord;
        readonly   CsLines:ShiftN;
        function   CsWords(Shift:ShiftN) return WordN;
        procedure  CsSetup;
        function   CsGetChar (Shift:ShiftN, Wn:WordN, Cn:CharN)
                       return CHARACTER;
    end Shifted-Lines;

    implementation
        Comp.ada      --recomputes the shifted lines
        Index.bss     --generates an index
        Tree.tc       --uses tree search
end CIRCULAR-SHIFTS

-----------------------------------------------------------------------

system KWIC
    provide KWIC

    module MC
            --Main module; acts as a driver for the others.
        provide procedure Kwic;
                exception Input-Error-Handler;
        require Input-Lines, Alph, Output-Lines
    end MC

-------------------------------------------------------------------------

    module INPUT
      --INPUT is the only module to know the input format.
      --Reads from the input, but calls other modules to store the lines.
        provide procedure Input Lines;
                exception Storage-Error-Handler;
        require Input-Error-Handler, Line

        implementation

            Term.ada        --input from terminal
            File.ada        --input from file
    end INPUT

-----------------------------------------------------------------------

    system INDEXGEN
        --Generates the circular shifts and stores them
        provide Alph, Ith, Shifted-Lines.{ShiftN, CsLines, CsWords,
                                             CsGetChar}
        require Line

        module ALPHABETIZER
          --Performs the sorting
          provide procedure Alph;
               procedure Ith (i:ShiftN) return ShiftN;
          require Shifted-Lines
          implementation
               Comp.ada  --recomputes alphabetization
               Core.ada  --stores sort in core
               SAM.for   --stores sort on seq. file
               ISAM.for  --stores sort on index seq. file
       end ALPHABETIZER

   composition INDEX = {ALPHABETIZER, CIRCULAR-SHIFTS}
   composition INDEX1 = {ALPHABETIZER.Core, CIRCULAR-SHIFTS.Comp}
   composition INDEX2 = {ALPHABETIZER.SAM, CIRCULAR-SHIFTS.tree}
   composition INDEX3 = {ALPHABETIZER.ISAM, CIRCULAR-SHIFTS.index}

   end INDEXGEN

------------------------------------------------------------------------

   module OUTPUT
        --Prints the KWIC-index; determines the output format.
        provide procedure Output-Lines;
        require Ith, Shifted-Lines

        implementation
             Term.ada     --Output to terminal
             File.ada     --Output to file
   end OUTPUT

composition KWIC = {MC, INPUT, LINE-STORAGE, INDEXGEN, OUTPUT}

composition KWICsmall = {MC, INPUT.Term, LINE-STORAGE.Core,
                             INDEXGEN.INDEX1, OUTPUT.Term}

composition KWICbig = {MC, INPUT.File, LINE-STORAGE.ISAM,
                           INDEXGEN.INDEX3, OUTPUT.FILE}

end KWIC
```

II. APPENDIX B -- MIL Syntax

II.1 Thomas' MIL Syntax

(1) <MIL_segment>::=**node** <node_name> {<synthesizes_part>}
{<must_inherit_part>}{<carrier_for_part>}
{<generates_locally_part>}{<successor_def_part>}

(2) <synthesizes part>::=**synthesizes** <resource_desc_list>

(3) <must_inherit_part>::=**must inherit** <resource_desc_list>

(4) <carrier_for part>::=**carrier for** <resource_desc_list>

(5) <generates_locally_part>::=**generates locally** <res_binding_list>

(6) <res_binding list>::=<res_binding>|<res_bindng><res_binding_list>

(7) <res_binding>::=<resource_desc> **using** <module_name>

(8) <successor_def_part>::=**has successors** <succ_desig_list>

(9) <succ desig_list>::=<succ_desig>|<succ_desig>,<succ_desig_list>

(10) <succ desig>::=<node_name>|<subsys_name>

(11) <successor_spec_list>::=<successor_spec> |
<successor_spec> <successor_spec_list>

(12) <successor_spec>::=**successor** <succ_desig>{<must_synthesize_part>}
{<inherits_part>}

(13) <must_synthesize_part>::=**must syntehsize** <resource_desc_list>

(14) <inherits_part>::=**inherits** <resource_desc_list>

(15) <resource_desc_list>::=<resource_desc> |
<resource_desc> <resource_desc_list>

(16) <resource_desc>::=<procedure_desc>|<cluster_desc>

(17) <procedure_desc>::=**proc** <identifier>

(18) <cluster_desc>::=**cluster** <identifier>
{**with ops** <operation_list>}

(19) <operation_list>::=<identifier>|<identifier>,<operation_list>

II.2 Cooprider's MIL Syntax

Subsystem Interconnection and Version Syntax

```
 (1) <subsystem>::=subsystem<name><connections><realization>
                   end<name>
 (2) <connections>::=[<provlist>][<reqlist>][<envlist>][<extrnlist>]
 (3) <provlist>::=provides<rsrc-name>{,<rsrc-name>}
 (4) <reqlist>::=requires <rsrc-name>{,<rsrc-name>}
 (5) <envlist>::=environment<rsrc-name>{,<rsrc-name>}
 (6) <extrnlist>::=external<subs-name>{,<subs-name}

 (7) <realization>::=realization<sellist><objlist>{<version>}
 (8) <sellist>::=select<subs-name>=<ver-name>{,<subs-name>=<ver-name>}
 (9) <objlist>::=concrete object<name>=<rule>{,<name>=<rule>}
(10) <version>::=version<name>[<sellist>][<objlist>][<vclists>]
                 end<name>
(11) <vclists>::={<version>}{<subsystem>}|[<cmps>][<rsrcs>][<dfrd>]
                 {<subsystem>}
(12) <cmps>::=component<object>{,<object>}
(13) <rsrcs>::=resources<object>{,<object>}
(14) <dfrd>::=deferred<object>{,<object>}
```

Construction Process Syntax

```
(15) <rule>::=<spec-rule>|<proc-rule>
(16) <spec-rule>::=file(<path>)|acquire(<rsrc-name>)|
                  deferred(<object>)
(17) <proc-rule>::=<proc-name>(<object>{,<object>:})
                   [with<string>]
(18) <object>::=<conc-name>|<rule>
```

NAMES:

```
(19) <name>::=subsystem name
(20) <rsrc-name>::=resource name
(21) <subs-name>::=name of a nested subsystem |
                   name of an external subsystem
(22) <ver-name>::=version name
(23) <proc-name>::=procedure name
(24) <path>::=file name | file and directory path name
(25) <conc-name>::=name of a concrete object
```

II.3 INTERCOL Syntax

```
(1) <INTERCOL-description> ::= [<default-language>]{<option-declaration>}
                  {<building-block>|,}{composition{<composition>|,}}
(2) <building-block> ::= <system-family>|<module-family>
(3) <system-family> ::= system<identifier> interface
                        <INTERCOL-description>
                        end [<identifier>]
(4) <composition> ::= <identifier> = <configuration>
(5) <configuration> ::= {{<element>|,}}[<extension>]|<element>
(6) <element> ::= <identifier>[<selection>][<extension>]
(7) <selection> ::= <implementation-selection>|<composition-selection>
(8) <implementation-selection> ::= .<identifier>[.<language>]
(9) <composition-selection> ::= [.<identifier>] [<refinement>]
(10) <refinement> ::= [.]({<element>|,})
(11) <module-family> ::= module<identifier> interface [<default-
                         language>]{<option-declaration>}
                         {implementation {<implementation|,}}
                         end [identifier]
(12) <implementation> ::= <program-family>|<subimplementation>
(13) <program-family> ::= <identifier> [.<language>] [<extension>]
(14) <subimplementation> ::= <identifier> = <program-family>
(15) <interface> ::= <provide-clause><require-clause> |
                     [<require-clause>][<provide-clause>]
(16) <provide-clause> ::= provide <resource-list>
(17) <require-clause> ::= require <resource-list>
(18) <resource-list> ::= {<resources>|,}
(19) <resources> ::= <resource-declaration>|see <character-string>
(20) <resource-declaration> ::= <compact-representation>|<specification>
(21) <compact-representation> ::= <single-resource> {.<single-resource>}
                                  [subresources]
(22) <single-resource> ::= [set] [#] <designator>
(23) <designator> ::= <identifier>|<character-string>
(24) <subresources> ::= [.] {<resource-list>}
(25) <option-declaration> ::= option {<alternative>|,}
(26) <alternative> ::= <identifier> [.{[any]{,|<alternative>}}]
(27) <extension> ::= {:<selector>}
(28) <selector> ::= <date>|<switch>
(29) <date> ::= [<|>] <number>|new [.<number>]
(30) <switch> ::= <identifier> {.<identifier>}
(31) <default-language> ::= language <language>
(32) <language> ::= <identifier>
```

<u>Syntax of the Resource-Specification Sublanguage</u>

The sublanguage is a part of Ada. Only the top-level productions and the ones that were modified are included. For additional information see the Ada manual.

(1) <specification> ::= <object-declaration>|<type-declaration>|<subtype-declaration>|<private-type-declaration>|<subprogram-declaration>|<entry-declaration>|<module-declaration>|<exception-declaration>
(2) <object-declaration> ::= <object-nature><identifier-list>:<type>;
(3) <object-nature> ::= **variable** | **readonly** | **constant**
(4) <identifier-list> ::= <identifier> {,<identifier>}
(5) <type> ::= <type-definition>|<type-mark>[<constraint>]
(6) <type-declaration> ::= **type**<identifier>[**is**<type-definition>];
(7) <subtype-declaration> ::= **subtype**<identifier>**is**<type-mark>
[<constraint>];
(8) <private-type-declaration> ::= [**restricted**] **type** <identifier>
is private [<type-definition>];
(9) <subprogram-declaration> ::= <subprogram-specification>;|<subprogram-nature><designator> **is**<generic-instantiation>;
(10) <subprogram-specification> ::= [**inline**|<generic-clause>]
<subprogram-nature><designator> [<formal-part>]
[**return** <typemark> [<constraint>]]
(11) <entry-declaration> ::= **entry** <identifier> [(<discrete-range>)]
[<formal-part>];
(12) <module-declaration> ::= <module-specification> | <module-nataure>
<identifier>[(<discrete-range>)]**is**<generic-instantiation>;
(13) <module-specification> ::= [<generic-clause>]<module-nature>
<identifier>[(<discrete-range>)] [**is**
{<specification>}{<representation-specification>}
end [<identifier>]];
(14) <exception-declaration> ::= **exception** <identifier-list>;

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.

[Balzer 73]
Balzer, R.M.
A Global View of Automatic Programming.
In Proceedings of the Third Joint Conference on Artificial Intelligence, pages 494-499. SRI International, aug, 1973.

[Balzer 79]
Balzer, R.M., and Goldman, N.
Principles of Good Software Specification and their Implications for Specification Language.
In Proceedings of the Specifications of Reliable Software Conference, pages 58-67. IEEE Press, 1979.

[Belady & Lehman 79]
Belady, L.A. and Lehman, M.M.
The Characteristics of Large Systems.
In Wegner, P., editor, Research Directions in Software Technology, chapter 1, pages 106-131. The Massachusetts Institute Technology Press, Cambridge, Mass., 1979.

[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.

[Dijkstra 76]
Dijkstra, E.
Structured Programming.
In Software Engineering, Concepts and Techniques. Litton Educational Publishing, Inc., 1976.
Originally appeared in a report on a conference sponsored by the NATO Science Committee, Rome, Italy, October 1969.

[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.

[Flon 75]
Flon, L.
Program Design With Abstract Data Types.
Technical Report, Carnegie-Mellon University, Computer Science Department, 1975.

[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. 76]
Haberman, A.N., Flon, L., and Cooprider, L.W.
Modularization and Hierarchy in a Family of Operating Systems.
Communications of the ACM, 19(5):266-272, May, 1976.

[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.

[Hammer 77]
Hammer, M., Howe, W., Kruskal, V., and Wladawsky, I.
A Very High Level Programming Language for Data Processing Applications.
Communications of the ACM, 20(11):832-840, November, 1977.

[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.

[Kernigham & Mashey 81]
Kernigham, B.W., and Mashey, J.R.
The Unix Programming Environment.
IEEE Computer Magazine, 14(4):12-24, April, 1981.

[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.

[Lehman 76]
Lehman, M.M.
Funnel- a Functional Data Channel.
IBM Technical Disclosure Bulletin, 1976.
Also Imperial College CCD Report 77/17, July 1977.

[Liskov & Zilles 74]
Liskov, B.H., and Zilles, S.N.
Programming With Abstract Data Types.
In Proceedings of the ACM SIGPLAN Notices Conference on Very High Level Languages, pages 50-59. SIGPLAN Notices , April, 1974.
Vol. 9, No. 4.

[Liskov & Zilles 75]
Liskov, B.H. and Zilles, S.N.
Specification Techniques for Data Abstractions.
IEEE Transactions On Software Engineering, SE-1:7-19, 1975.

[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.

[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.

[Morrissey 79]
Morrissey, J.H., and Wu, L.S.-Y.
Software Engineering ... An Economic Perspective.
In Proceedings of the Fourth Intl. Conference on Software Engineering, pages 412-422. IEEE, 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 72a]
Parnas, D.L.
On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the ACM, 15(12):1053-1058, December, 1972.

[Parnas 72b]
Parnas, D.L.
A Technique for Software Module Specification With Examples.
Communications of the ACM, 15(5):330-36, May, 1972.

[Parnas 76]
Parnas, D.L.
On the Design and Development of Program Families.
IEEE Transactions On Software Engineering, SE-2(1):1-9, March, 1976.

[Parnas 78]
Parnas, D.L.
Designing Software for Ease of Extension and Contraction.
In Proceedings of the Third Intl. Conference on Software Engineering, pages 264-277. IEEE Press, March, 1978.

[Parnas 79]
Parnas, D.L.
Designing Software for Ease of Extension and Contraction.
IEEE Transactions On Software Engineering, SE-5(2):128-138, March, 1979.

[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.

[Prywes et.al. 79]
Prywes, N.S., Pnueli, A., and Shastry, S.
Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development.
ACM Transactions on Programming Languages and Systems, 1(2):196-217, October, 1979.

[Prywes 77a]
Prywes, N.S.
Automatic Generation of Computer Programs.
In Proceedings of the National Computer Conference, pages 679-689. AFIPS Press, 1977.

[Prywes 77b]
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.

[Ritchie & Thomson 74]
Ritchie, D.M. and Thompson, K.
The UNIX Time-Sharing System.
Communications of the ACM, 17(7):365-375, July, 1974.

[Ritchie 79]
Ritchie, D.M.
The Evolution of the UNIX Time-Sharing System.
In _Procceedings of the Symposium on Language Design and Programming Methodology_. ACM SIGPLAN, Sydney, Australia, 1979.

[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.

[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.

[Wirth 71]
Wirth, N.
Program Development by Stepwise Refinement.
Communications of the ACM, 14(4):221-27, April, 1971.

[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.