Reprinted with permission from *Journal of Systems and Software*, Volume 6, Number 4, November 1986, pages 307-334.

Module Interconnection Languages

Ruben Prieto-Diaz

Computer Science Laboratory, GTE Laboratories Inc., Waltham, Massachusetts

James M. Neighbors*

Department of Information and Computer Science, University of California, Irvine

Module interconnection languages are considered essential tools in the development of large software systems. Current software development practice follows the principle of the recursive decomposition of larger problems that can be grasped, understood, and handled by specialized and usually independent teams of software engineers. After teams succeed in designing and coding their respective subsystems, they are faced with different but usually more difficult issues; how to integrate independently developed subsystems or modules into the originally planned complete system. Module interconnection languages (MILs) provide formal grammar constructs for describing the global structure of a software system and for deciding the various module interconnection specifications required for its complete assembly. Automatic processing of these formal descriptions results in a verification of system integrity and intermodular compatibility. This paper is a survey of MILs that are specifically designed to support module interconnection and includes brief descriptions of some software development systems that support module interconnection.

1. INTRODUCTION

Module interconnection languages are considered essential tools in the development of large software systems. Current software development practice follows the principle of the recursive decomposition of larger problems into smaller problems that can be grasped, understood, and handled by specialized and usually independent teams of software engineers.

After teams succeed in designing and coding their respective subsystems, they are faced with different but usually more difficult issues; e.g., how to integrate independently developed subsystems or modules into the originally planned complete system. Manual integration has been the standard practice with increasing support from automated environments. The current state of the art in software development environments is due mainly to the pioneering work on module interconnection language of the late 1970s. Module interconnection languages (MILs) provide formal grammar constructs for deciding the various module interconnection specifications required to assemble a complete software system. An MIL code listing is a formal description of the global structure of a software system. Automatic processing of these formal descriptions results in a verification of system integrity and intermodular compatibility.

An MIL can be considered a structural design language because it states what the system modules are and how they fit together to implement the system's function. This is architectural design information. MILs are *not* concerned with what the system does (specification information), *how* the major parts of the system are embedded into the organization (analysis information), or how the individual modules implement their function (detailed design information).

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

The first module interconnection language, MIL75, was developed by DeRemer and Kron [14]. They established the basic ideas and concepts of module inter-

*This work has been partially supported by NSF grant MCS-81-03718.

Address correspondence to Ruben Prieto-Diaz, Software Engineering Environments, GTE Laboratories, Inc., 40 Sylvan Rd., Waltham, MA 02254.

The Journal of Systems and Software 6, 307–334 (1986)

0164–1212/86/$3.50

connection 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 rquired for "knitting" those modules together.

Three MILs [48, 12, 50] were developed after MIL75, Thomas' MIL [48] Cooprider's MIL [12], and INTERCOL [50], each one adding new ideas and features to MIL75 but, essentially based on DeRemer and Kron's original concepts. Thomas developed a module interconnection notation and discussed a possible module interconnection processor; Cooprider expanded the basic ideas of MIL75 to introduce a version control facility and a software control facility; and Tichy developed INTERCOL, a MIL that integrates some of Cooprider's features with control of system families and with independent compilation of modules.

There exist several languages, software development tools, and operating systems that in one way or another provide module interconnection mechanisms. To describe every tool, system, or methodology that supports some kind of module interconnection is beyond the scope of this paper. The scope of this paper is to survey the languages called *M*odule *I*nterconnection *L*anguages that are specifically designed to support module interconnection and to include brief descriptions of some software development systems that support module interconnection to provide a frame of reference for comparisons.

The goal of this survey is to acquaint the reader familiar with the problems of programming and maintaining large software systems with a class of tools designed to describe how a large system "fits together." In particular, we will focus on MILs designed to specify the structure of a system, not its behavior. We will not deal largely with how the MILs perform their functions in order to treat the issue of *what* they do in more detail. The interested reader will find more detail in the referenced literature.

1.1 Current Research

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

Researchers in Software Engineering view the problem as a design problem and approach the problem from the point of view of finding a design notation that 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 that might occur. The SARA system [9, 10] has adopted this structural modeling and Ada®[1] based real-time system designs use this approach for integrating multitasking modules. 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 that must be implemented, a consistency model determines a collection of specific versions and implementations of modules that can be shown to implement the system [34, 35].

For the Artificial Intelligence researcher, the interconnection problem manifests itself as a problem in automatic programming. In this context, "knowledge about programming" or "knowledge aout 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 that 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 that might suffice for a given goal (plan), and a description of 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, which is the interconnection problem. The Transformational Implementation system [3] and the Programmer's Apprentice system [41, 42] are two systems that take this approach.

The point of view taken in this survey is the Software Engineering perspective. The points of view of the other perspectives are very important and each deserve a complete in-depth study. Since each of these views deal with similar interconnection information, it is impor-

[1] Ada® is a registered trademark of the Department of Defense, Ada Joint Program Office.

tant that a researcher using one perspective understand the focuses of the other perspectives including information manipulated and the operations provided.

1.2 Plan of Presentation

This survey is divided into three main sections. First, MIL concepts and ideas are presented and a domain of discourse is established. The next section is a detailed presentation of the four classical MILs: MIL 75, Thomas' MIL, Cooprider's MIL, and Intercol. A common example is implemented in all four MILs to compare and contrast their features. The last section is a brief description of some software development systems that support module interconnection and how their features relate to the classical MILs.

2. MIL Concepts and Ideas

Modularity is a well-established concept that has been used in engineering and managerial disciplines for many years to break up the work of a big project into controllable units. In programming, the main idea is to separate the behavior of the program from the details of each component, thus reducing the complexity of the programming problem. Each of the subprograms can then be considered in turn, in isolation from each other and from the program skeleton in which they are embedded.

Some of these ideas go as far back as the concept of mathematical function or even to earlier times. Even though the key words "modularization" or "module" did not become widely used until the early 1970s, the original work on structured programming and hierarchical system decomposition provided the conceptual background for the development of module interconnection languages (MILs) in the late 1970s.

The concept of MILs came about as part of the conceptual separation between programming-in-the-large (PL) and programming-in-the-small (PS). PS is concerned with building programs and has been greatly developed to include the new techniques of structured programming, top-down design, stepwise refinement, and others. Many of the widely accepted languages (ALGOL, Pascal, COBOL, etc.) have been designed to aid programming-in-the-small and have contributed towards the effort to make programming a science [23]. The system life cycle phases of detailed design and implementation primarily use PS notations. These notations focus on how a particular part (module) of a system performs its function.

PL, on the other hand, is concerned with building systems. PL notations are primarily used in the architectural design phase of system construction and concentrate on how the system modules cooperate (through calls and data sharing) and which functions each module provides. We refer to a language concerned with the data and control flow interconnections between a collection of modules as a *L*anguage for *P*rogramming in the *L*arge (LPL). An MIL is an LPL with a formal machine-processable syntax (i.e., not natural language or graphical diagram) that provides a means for the designer of a large system to represent the overall system structure in a concise, precise, and verifiable form. As in conventional languages for PS, an MIL assumes the existence of a language processor for coupling programs (i.e., system structure descriptions) expressed in MIL syntax.

MILs are very effective but limited tools to aid during the software life-cycle. A system must be analyzed, evaluated, and designed first by means of current methods and techniques. Once a system structure is determined, it may be coded in an MIL to be checked and verified for completeness and inconsistencies. MIL code must be maintained during implementation and then used for high-level maintenance during system operation and enhancement.

The main concepts of MILs are:

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

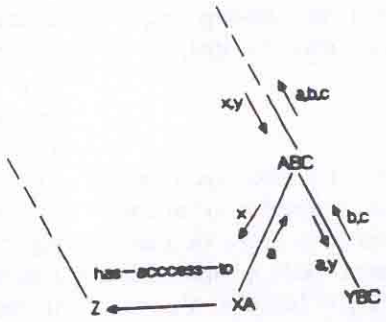
An MIL usually serves as a *Project Management Tool* by encouraging structuring before the start of detailed programming and as a *Support Tool* for the design process by capturing overall program structure and being able to verify system integrity before implementation. An MIL could also provide some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

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

All resources are ultimately provided by *modules*, thus modules are units that provide resources and that require some set of resources. The syntax primitives of

```
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
```

MIL source code



Graphical representation

Figure 1. MIL description of a module.

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

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

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

In most of the module interconnection schemes we shall examine, the PL information is in the form of an MIL and the PS information is in the form of a regular programming language. The packaging of this information differs between different schemes (Figure 2). At one side of the spectrum a system is defined as a collection of modules each of which contains MIL and PS information with no central description of the system other than the list of modules that compose it. At the other end of the spectrum, the modules that 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 its constituent parts match. No programming language (PS level) information is necessary to perform this compilation. There are some functions that are not considered to belong to the domain of MILs. These functions were stated by DeRemer and Kron [14] and by Thomas [48] in order to make a clear distinction between an 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.

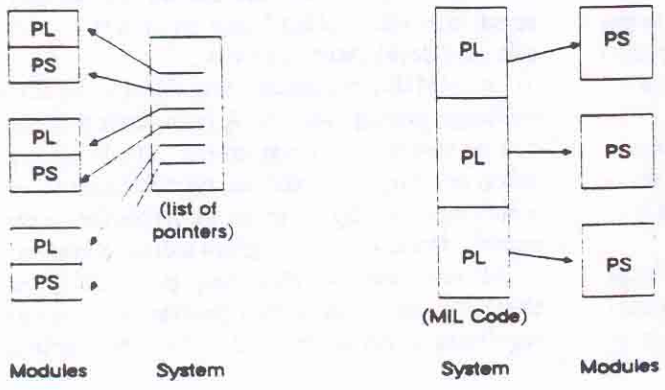


Figure 2. Two alternatives to describe system structure using MILs.

The functions MILs and their respective processors should not attempt are:

1. *Loading:* An MIL should leave this function to a "subsystem loading language" or to other facilities within the software development environment.
2. *Functional System Specification:* An MIL only shows the static structure of a piece of software and should not specify the nature of its resources. This task should be assigned to other subsystems.
3. *Type Specification:* An 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 syntactically checked by an MIL processor for type consistency throughout the system but an MIL cannot check for the validity of their specifications. For example, the decision to declare **real** y in a program is a design decision that follows a type specification while the statement **real** y in MIL code is processed as a type checking statement only. An MIL, however, is used to display the structure of a system to help in validating specifications.
4. *Embedded Link-Edit Instructions:* These operations should be left for another subsystem within the development environment such as the operating system or a separate command language.

The current tendency in MIL development is to keep the domain of MILs well defined so that stand-alone MILs can be developed and then integrated as part of a software development environment such as in Gandalf [21, 24].

Approaches such as C/MESA of the MESA System [29] and *External Structure* of the ADAPT system [1] conform to the current tendency but are not as general since they are restricted to modules coded in a single programming language.

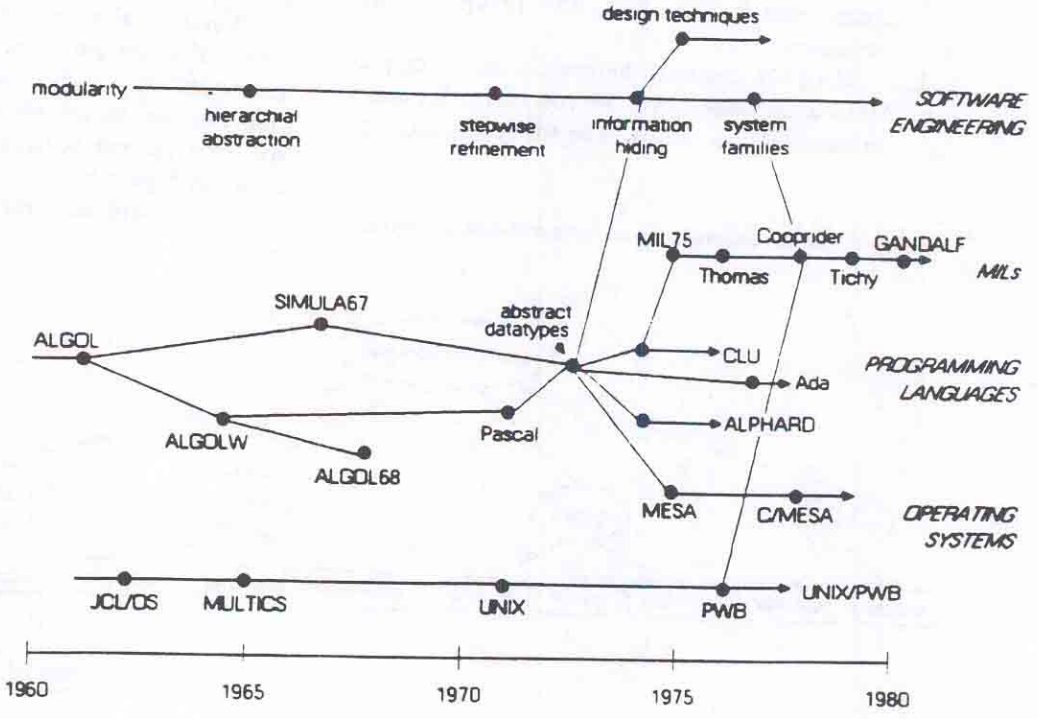
More recent programming environments provide tools that support module interconnection along with version control mechanisms and other software development aids (i.e., PWB, MESA, CDL2, Gandalf, Pamela, Arcadia, etc.). Some of the module interconnection tools integrated in these systems are implementations of MILs (i.e., MESA's C/MESA, SARA's MID, Gandalf's INTERCOL, DREAM's DDN) while others are collections of specialized tools (i.e., PWB, PROTEL, CDL2). Some of these tools and environments are described in more detail in Section 4.

3. MODULE INTERCONNECTION LANGUAGES

There are four stand-alone MILs developed to date and reported in the literature: MIL75 [14], Thomas' MIL [48], Cooprider's MIL [12], and INTERCOL [50].

The evolution of MILs along time is shown in Figure 3. MIL75 and Thomas' MIL have a strong modular

Figure 3. Graphic view of MIL evolution.



language origin and a tendency to use software engineering techniques. Cooprider's MIL and Intercol, although sharing the same origins with their predecessors, go further by integrating techniques from software engineering and tools from powerful operating systems. The graph partially represents the citations made by papers in the different areas at the given times and serves to show that the problem addressed by MILs are not confined to a single area.

DeRemer and Kron developed the first module interconnection language, MIL75 [14]. They established the basic ideas and concepts of module interconnection. MIL75 is a language for programming-in-the-large (LPL) that gives the systems designer a tool to design and, to a certain extent, build a complete system out of modules that do not have to be completely coded and tested, just properly specified. For each module, the designer must specify the resources provided and required. The type of the resources must also be specified. Details about the internal operations of the modules are not required. MIL75 processes all these specifications while doing consistency checking resulting in an accurate recording of the overall solution structure.

Thomas [48] developed a module interconnection notation and discussed a possible module interconnection processor. He proposed a formal model based on the separation of compiling, binding, and linking that allows for flexible bindings and also provides a notation to incorporate his MIL into a programming environment. Besides the flexible binding scheme which is his main contribution to MILs, Thomas presented, through his formal model, the basis for practical MIL implementation.

Cooprider [12], expands the basic ideas of the previous MILs to introduce a version control facility and a software construction facility. The former facility can recognize the different instantiations (versions) of an interconnection network and know how they are hierarchically integrated while the latter facility can construct a complete software system from a functional description of the construction process. Resources and source files are combined according to construction rules, and 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 database processor (similar to the system described by Bratman and Court [8]) supporting an interactive system construction environment.

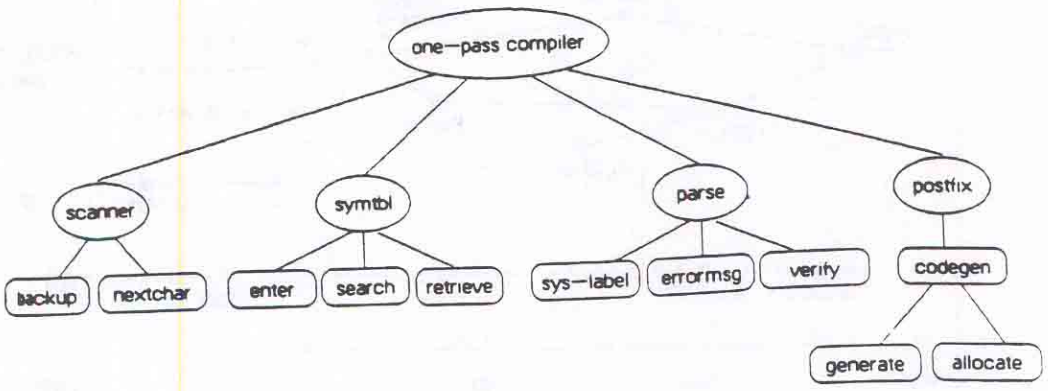
INTERCOL was developed by Tichy in 1979 [49, 50]. In addition to the features of Cooprider's MIL, INTERCOL supports separate compilation of modules and/or subsystems, and control of system families. INTERCOL is intended to be an integrated software development and maintenance environment that supports communication and cooperation among programmers. Gandalf has integrated INTERCOL as its tool for system version description and generation.

3.1 MIL75

MIL75 is based on the concept that any system structure has a graphical representation in the form of an inverted tree with nodes being the modules and the edges their different hierarchical relationships. This graphical relationship of a system is an implicit prerequisite to using MIL75. The methods proposed in [36, 47, 52] 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 accessiblity as well as the resources required and provided. An inverted tree structure of a one-pass compiler is shown in Figure 4.

Once a graphical structure for a system is obtained,

Figure 4. Graphical system tree for a one-pass compiler.



it is programmed in MIL75 where the code consists of the description of the modules in each node. The code is processed 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 of items that are required to establish system structure:

1. *Resources:* Atomic elements that 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 (listed below) are based around the inverted tree model described above and form the minimum set required by MIL75 to be able to do module interconnection:

1. *External scope relations:* These relations define the scope of definitions of module or subsystem names thus helping to impose the overall system structure called here the *system tree.* These external scope definitions consist of the descriptions of nodes and their respective interfaces written in MIL75 code.
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 (e.g., from children to parent) and derived (e.g., from parent to children) for each node pursuing a top-down approach. Resources originated in other nodes, not being direct ancestors or successors, are *accessed* resources.
3. *The relationship among the resources of sibling modules:* The channels for flow of resources among siblings are determined by the parent. These accessiblity channels or links among a set of siblings may form any directed graph. Access rights are not transitive and the children of a node are invisible to its siblings. This relationship limits resources accessibility to modules lying at the same hierarchical level.
4. *The relationship of accessibility of resources of modules at different hierarchical levels:* Unless explicitly stated by a parent, children inherit by default all access rights granted to their parents. Parents have unconditional access to their children but not to their grandchildren or lower descendants.
5. *The relationship between modules and the origin and usage of resources:* For each module, a MIL75 program must include two statements: the *statement of origin* listing the resources defined in that module and, the *statement of usage* listing separately the derived resources and all other resources obtained through siblings or inherited access.

Establishing relationships 1–5 is what MIL75 coding is all about. After the system designer has coded these relationships, the MIL75 "compiler" checks that actual usage of resources by a given module agrees with the 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 corrresponds to binding (at compile time). This binding is what establishes the "module interconnection structure" shown in Figure 5. A complete MIL75 program consists of a series of statements expressing the different relationships (1, . . . , 5 above) between resources and modules of a structured (nodal) representation of a system.

The MIL75 code that describes the module interconnection structure for the one-pass compiler of Figure 5 is shown in Figure 6. In order to minimize MIL coding effort, anyone doing programming-in-the-large should start describing the system from the beginning of the design process rather than wait for the design to be completed and then proceed to describe it.

Differences from the other MILs. MIL75 is oriented around a structured (oriented tree) representation of a system thus shifting some of the work back to the system designer. The MIL75 compiler takes the complete description of the system where design decisions like proper abstraction, functional decomposition, and modularization have already been made by the system designer. The system designer must also establish the accessibility and provision of resources among modules.

The main contribution of MIL75 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 system modularization or simply an inconsistency

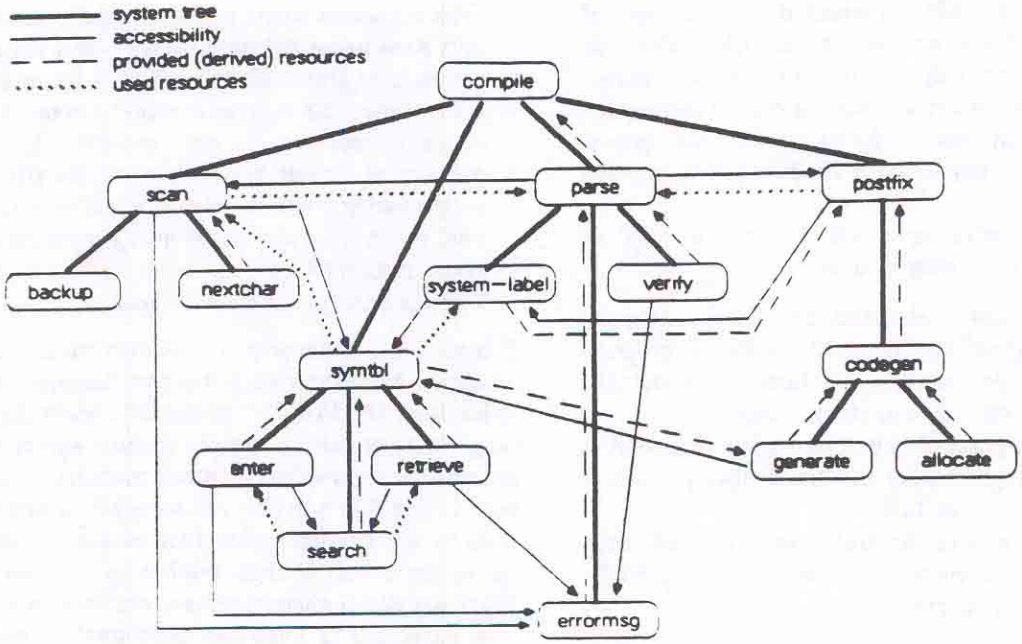


Figure 5. The module interconnection structure.

in the flow of resources. In the later case, the fix is relatively easy and requires the recompilation of one or a 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 that a MIL should have the capability to support modules programmed in different languages but they have not established this capability in MIL75. Last but not least, MIL75 may be seen as an isolated tool used only to show how an MIL should work but not how it would be integrated into a software development environment. Integration is essential to reduce maintenance effort. Maintenance of two separate source listings for one project may become a problem. An MIL should be integrated into a software development environment where updates in programming-in-the-large can be automatically indicated in programming-in-the-small and vice versa. Thomas tried to establish the mechanisms to integrate his MIL into a programming system. This integration is required in order to use and evaluate MILs.

Experience to date. Although DeRemer and Kron do not mention any particular implementation of MIL75, they implied the existence, at least at the prototype level, of an MIL75 processor to test some of the examples. The objective was to test the concepts of module interconnection, but not to use MIL75 in a production environment or to integrate it into a software development system.

3.2 Thomas' MIL

The objective in Thomas' dissertation [48] was to propose an MIL that would be a complement to CLU/ALPHARD-like languages (i.e., languages that support data abstraction) and that would incorporate new ideas for the future development of MILs.

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

This scheme allows a software system to be represented (in MIL code) as a "finite directed graph G with no simple cycles and where S is a start node in G and all nodes in G are reachable from S." An instance of this graph definiton may be the inverted tree module interconnection structure of MIL75. Thomas proves that static checking will not be affected by the addition

```
system compile
author John Smith
date   2/25/82
provides compiler
consists of
  root module
    originates compiler
  subsystem scan
    has access to symtbl, errormsg
    consists of
      root module
        originates scanner
        uses derived nextchar_funct
        uses nonderived symtbl
      subsystem nextchar
      must provide nextchar_funct /* fetches next char */
      subsystem backup
      provides backup_proc /* backs up to previous char */
  subsystem parse
    must provide parser
    has access to scan, postfix
    consists of
      root module
        originates parser
        uses derived verify_funct, errormsg_proc
        uses non derived scan, postfix
      subsystem verify
        must provide verify_funct /* checks for valid tokens */
        has access to errormsg
      subsystem errormsg
        must provide errormsg_proc /*displays error type and nbr*/
      subsystem system-label
        provides system_label_proc /* creates system labels */
        has access to symtbl
        uses non derived symtbl
  subsystem symbtbl
    consists of
      root module
        originates symbtbl_proc
        uses derived enter_proc, get_proc, lookup_proc
      subsystem enter /* enters a symbol in symbol table */
        must provide enter_proc
        has access to search, errormsg
        uses non derived search
      subsystem retrieve /* gets an entry from symbol table */
        must provide get_proc
        has access to search, errormsg
        uses non derived search
      subsystem search /* look for a required symbol */
        must provide lookup_proc
  subsystem postfix
    has access to  system_label
    consists of
      root module
        originates postfix_generator /*convert to post_fix notation*/
        uses derived codegenerator
        uses non derived system_label
      subsystem codegen
        must provide codegenerator /* generates machine code */
        consists of
          root module
            originates codegenerator
            uses derived register_allocation, generate_instruction
          subsystem allocate
            must provide register_allocation
          subsystem generate /* generates address and operands
                                for machine instruction */
            must provide generate_instruction
            has access to symtbl
            uses non derived symtbl.
```

Figure 6. A simple one pass compiler described in MIL75.

of cycles but that binding may become an intractable problem in some cases. His proof is based on 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 (i.e., cycles). Expensive dynamic linkage must be used for these cases instead. Of course, Thomas obtains interconnection flexibility by going from a pure oriented tree structure

of a system to an oriented tree with cycles at the price of sometimes not being able to do the binding.

The "universe of discourse" of Thomas' MIL is *names* that fall into four classes: Resources, Modules, Nodes, and Subsystems.

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

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

Nodes are descriptive units (in MIL code) that establish environments for the modules by binding resource names to modules. Nodes are the basic entity for programming-in-the-large just as a module description is in MIL75. So a node specifies the set of modules attached to it and the interconnection between the node and other nodes of the system. There are four main operations a node can apply to 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 the MIL75 **provide, has-access-to,** and **consist-of,** respectively.
4. *Has-successor:* determines the set of nodes that provides resources to this node, or in MIL75 terms, the successors are the children of a node that generate "derived" resources to their parent.

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

In [48], complete syntax description and examples of this MIL are presented for further reference. The one-pass compiler from the previous example coded in Thomas MIL is shown in Figure 7. Notice the layered decomposition of the system in contrast with the rigid hierarchy of MIL75. Each node description is self-contained (i.e., capitalizes on the data abstraction principle). It describes its composition, and the resources provided and required to interact with other nodes. Notice also that nodes may be arranged in any order in the proper listing. The introduction of the **using** construct in the syntax allows Thomas' MIL to specify which particular implementation of a module or function is required. If a user were to design a software system using Thomas' MIL as a development tool, a structured design methodology should be followed to obtain an oriented tree structure of the system just as the "system tree" of MIL75 is obtained. The user would then define the nodes in MIL constructs by carefully analyzing which modules could be encapsulated in a subsystem so that a node structure is obtained which describes the whole system structure in an LPL. This is analogous to the "packaging" activity of structured design [36] or the informaton hiding principle of Parnas [37]. This is a more flexible way to build the rigid "resource augmented" and "access augmented" system trees of MIL75.

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

The next step, in contrast with MIL75, would be to code the individual modules and compile each one separately. Finally, the MIL processor would be called to do the binding and perform the required module interconnections; i.e., to change all indirect references to direct connections. The MIL75 compiler instead establishes the "usage links" (bindings) at an LPL level without need for module coding.

Difference from the other MILs. Thomas' MIL performs the module interconnection after module compilation allowing more flexibility to the designer at the type-check stage but at the same time forcing the derivation 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. A change in MIL75 code, in contrast, usually requires a full recompilation. This pay-off will be increased if the MIL were to be integrated into a system development environment as Thomas proposes. Thomas' MIL is restricted by binding the interconnection to the compile/link paradigm. Cooprider and Tichy succeed in freeing their MILs from this restriction and in integrating their MILs into working systems development environments.

Experience to date. Thomas' work was only a discussion of a possible MIL processor and it was not implemented. It is, however, a valuable work that provided some basic proofs on MIL structures and established certain ideas for future MILs.

```
node compile
  synthesizes proc compiler
  generates locally proc compiler using NIP_COMP
  has successors scan, parse, symtbl, postfix
    successor scan
      must synthesizes proc scanner
      inherit proc symbtbl_funct, errormsg_proc
    successor parse
      must synthesizes proc parse
      inherit proc scanner, postfix_generator
    successor symtbl
      must synthesize cluster symtbl_proc
                              with ops enter, retrieve, search
    successor postfix
      must synthesize proc postfix_generator

node scan
  synthesizes proc scanner
  must inherit proc symtbl_proc, errormsg_proc
  generates locally cluster scanner using SCAN02
              proc nextchar_funct using NXTCH
              proc backup_proc using BCKP

node parse
  synthesizes proc parser
  must inherit proc scanner, postfix_generator
  generates locally proc parser using PARSE.NPP
  has successors system-label, errormsg, verify
    successor system-label
      must synthesize proc system-label_proc
      inherits proc symtbl_proc
    successor errormsg
      must synthesize proc errormsg_proc
    successor verify
      must synthesize proc verify_funct
      inherits proc errormsg_proc

node symtbl
  synthesizes proc symtbl_proc
  must inherit proc errormsg_proc
  generates locally cluster symtbl_proc using SYMBL
                      proc enter_proc using ENT01
                      proc get_proc using RET00
  has successors search
    successor search
      must synthesize proc lookup_proc
      .
node postfix
  synthesizes proc postfix_generator
  must inherit proc system-label_proc
  generates locally proc postfix_generator using POSTFIXGEN.NIP
  has successors codegen
    successor codegen
      must synthesize proc codegenerator

node codegen
  synthesizes proc codegenerator
  must inherit proc symtbl_proc
  generates locally proc cluster codegenerator using CODEG03
                proc register_allocator using REGALLOC
                proc generate_instruction using GEN07
  inherits proc symtbl_function

node system-label
synthesize proc system-label_proc
must inherit proc symtbl
generates locally proc system-label_proc using SYSLBL

node errormsg
synthesize proc errormsg_proc
generates locally proc errormsg_proc using ERRMSG

node verify
synthesize proc verify_func
generates locally proc verify_func using VRFY

node search
synthesize proc lookup_proc
generates locally proc lookup_proc using LOOKUP02
```

Figure 7. A one pass compiler coded in Thomas' MIL.

3.3 Cooprider's MIL

The objective in Cooprider's work [12] is to propose a system, that to some extent, would bridge the gap between software design and software construction. He developed a representation for software systems that integrates an MIL, a version control facility, and a software construction facility. His emphasis is on the later two facilities but he succeeded in adding some innovations to the work of DeRemer & Kron and Thomas.

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

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

There are three interconnection mechanisms in this MIL:

1. *Nesting:* The provider can be nested directly within a requirer. This mechanism is similar to the flow of resources from children to parent in the resource augmented tree of MIL75.
2. *Explicit Reference:* The provider can be named by an external clause in the requirer. This case is analogous to the accessibility channels for resources among sibling modules of MIL75.
3. *Environment Definition:* The provider can be named by a subsystem that encloses the requiring subsystem. This mechanism is the same environment described in Thomas' MIL and similar to the flow of resources from parent to children in the resources augmented tree of MIL75.

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

There are three operators used in the construction system:

1. *File:* Used to point to a specific file name indicated by a *path* (full directory path) enclosed in ⟨ ⟩ brackets. This path may be empty, thus showing the file name only.
2. *Acquire:* converts a resource from another subsystem into a concrete object.
3. *Deferred:* retrieves all objects that have been implicitly associated with the parameter object. This operator is used when separately compiled subroutine bodies are linked and their external procedure declarations made effective.

The example below illustrates the use of the above operators.

Example:

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

The version control system. The objective of this system is to make different system versions share the same interconnection structure so that duplication of identical information is prevented and modification sites are centralized. This approach is better than copying system descriptions that would require modifications to each copy for any small alteration performed to a component subsystem.

The syntax for this system consists of two parts: the *realization* section and the *version* section. The realization section contains all the information pertinent to the tangible form of a subsystem while a version is an instantiation of a subsystem or a group of such instantiations. There are several combinations of the syntactic constructs that can be used to describe a subsystem realization. The example that follows 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⟩)end FORTRAN
   version Pascal resources file(⟨PascalCarefulHash⟩)end Pascal
   version ALGOL resources file(⟨ALGOLCarefulHash⟩)end ALGOL
 end Careful end HASH
```

Our one pass compiler coded in Cooprider's MIL is shown in Figure 8. Emphasis is on the interconnection mechanism to compare with MIL75 and with Thomas' MIL. Hypothetical realizations are included in the first two subsystems (scan and parse) to illustrate how a complete program might look in Cooprider's MIL. The remaining subsystem realizations were omitted for simplicity and only the names of the corresponding provided resources are listed. The approach to structuring a system is very similar to that used by Thomas. Cooprider introduces optional nesting as shown in the compiler and symtbl subsystems of the example. Optional nesting provides more flexibility to the system design and is an effective tool for modular design based on information hiding.

In contrast with the two previous MILs, the language developed by Cooprider could be seen as an extended MIL that also supports system construction not only system design. If a user were to design and construct a software system using this MIL as a development tool, a similar process would be followed as if using MIL75 or Thomas' MIL; i.e., a structured design methodology. This process would be, in contrast with the previous MILs, carried on interactively with the aid of a database system where system integrity would be verified. Availability of implementation information at the MIL level significantly reduces maintenance effort and increases reliability.

With this tool, construction information could be specified and verified during the design phase so that the end product would not be only a structured description of what steps to follow to obtain such a system. Module coding could be done separately and/or in parallel with the whole system design. The largest gain in using Cooprider's system would be, by far, during the evolution of the software product throughout its entire operational life.

Differences from the other MILs. It is difficult to compare Cooprider's MIL with the previous two because of its language extensions. Judging the structural flexibility offered to the system design, however, Cooprider's MIL could be placed in the middle point between the very rigid MIL75 and the highly decoupled Thomas' MIL. The module interconnection mechanisms of Cooprider's MIL could be considered as a synthesis of both. That is, most of their advantages such as flexibility in the interconnection structure, easy syntax and notation and static binding were integrated in this MIL. There are restrictions on the flow of resources in Cooprider's MIL as there are in MIL75, but they are not as stringent. A subsystem here **provides** and **requires** resources in a way that is similar to scope rules in structured programming languages, while in a module in MIL75, **derived** and **accessed** resources must also be specified to indicate flow among parent–offspring or among siblings, respectively. This reduction in the complexity of resource flow is due to the use of a database processor instead of a compiler. This is the major contribution of Cooprider's work. The database 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 database has no knowledge of the various versions. The realization description requires excessive detail; the designer must give explicit construction rules for all components and configurations and program all the modification policies by hand. Moreover, the database processor does not support control for concurrent actions (i.e., two programmers modifying the same file at the same time).

Experience to date. Several parts of this system have been implemented. The implemented components were tested in a laboratory environment with a specific but small test case: 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

```
subsystem compile provides compiler
                  requires scanner, symtbl_proc, parser,
                           postfix_generator

  subsystem scan provides scanner
                 requires backup_proc, nextchar_funct
                 external symtbl, errormsg, parse

    subsystem backup provides backup_funct
      realization
        version NIP-language
          version PL/1 concrete object BCKP = PL
                    (file(<BCKP.service>)) end PL/1
        end NIP-language
    end backup

    subsystem nextchar provides nextchar_funct
      realization
        version NIP-language
          version PL/1 concrete object NEXTCH=PL
                    (file(<NXTCH.source>)) end PL/1
        end NIP-language
    end nextchar

    realization
       version NIP_language
         version PL/1 concrete object SCAN02=PL
                   (file(<scan02.source>)) end PL/1
         version Pascal concrete object SCAN03-Pasc
                    (file(<scan03.Pscl>)) end Pascal
       end NIP_language
    end scan

  subsystem parse provides parser

    requires system-label_proc, errormsg_proc, verify_funct
    external scan, postfix
    realization
      version NIP-language
        version PL/1 concrete object PARSE.NIP=PL
                  (file(<PARSE.source>)) end PL/1
      end NIP-language
  end parse

  subsystem postfix provides postfix_generator
            requires codegenerator external system-label
    realization .. (creates POSTFIXGEN.NIP)
  end postfix
```

Figure 8. Partial Cooprider's MIL code for a simple one pass compiler.

development of the Gandalf System Generator Facility [24].

3.4 Intercol

Tichy's work at the software development environment level has three objectives:

1. A Module Interconnection Language (INTERCOL) capable of representing multiple versions and configurations written in multiple programming languages.
2. An Interface Control System that automatically verifies interface consistency among separately developed software components.
3. A Version Control System similar to the one proposed by Cooprider but with the advantage that in this case the system determines which version of which component should be used to form a particular version of a particular configuration instead of relying on a detailed set of construction commands issued by the designer as in Cooprider's MIL.

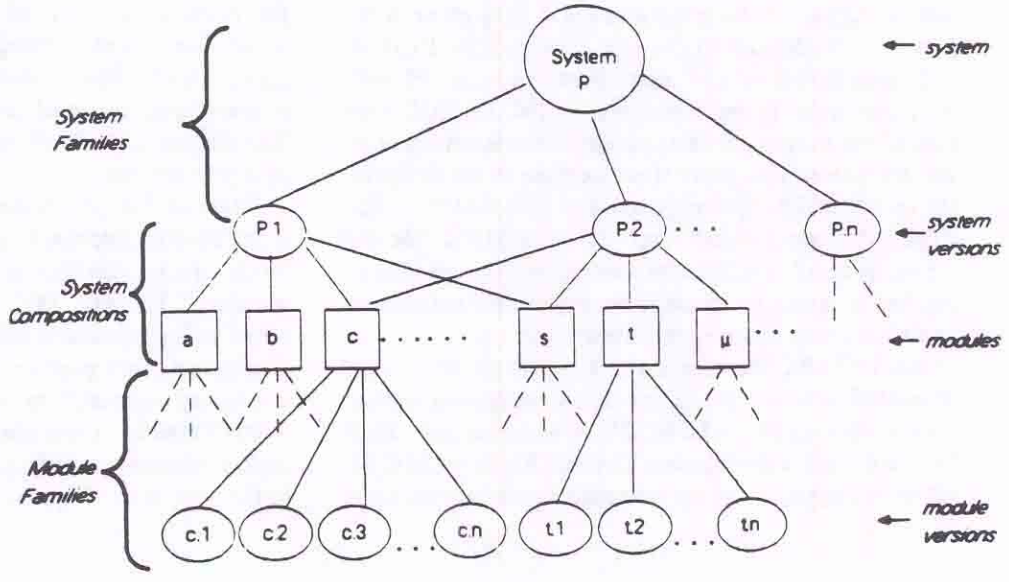
A description in INTERCOL is a sequence of module and system families followed by a set of compositions. A member of a *module family* is a version of a module, and a member of a *system family* is a version of a system (Figure 9). The former may be one of a set of different module *implementations* for different environments or in different languages, or may be one of a set of different module *revisions*, or can also be a *derived version*. The latter may be a member of a set of different system *configurations* or of a different *derived composition*.

Each one of the above families has an *interface*. An interface consists of programmed entities called *resources*. A resource in INTERCOL has the same meaning as a resource in the previous MILs; they are the units of flow among modules and/or among systems. All members of a particular module or system

```
subsystem symtbl provides symtbl-proc
                  requires enter-proc, lookup-proc, get-proc
                  external errormsg
  subsystem enter provides enter-proc
                   external search, errormsg
    realization .. (creates ENT01)
  end enter
  subsystem search provides lookup-proc
    realization .. (creates LOOKUP02)
  end search
  subsystem retrieve provides get-proc
                      external search, errormsg
    realization .. (creates RET00)
  end retrieve
  realization .. (creates SYMBL)
end symtbl

subsystem system-label provides system-label-proc
                       external symtbl
  realization ..(creates SYSLBL)
end system-label

subsystem errormsg provides errormsg-proc
  realization ..(creates ERRMST)
end errormsg

subsystem verify provides verify-funct
                  external errormsg
  realization .. (creates VRFY)
end verify

subsystem codegen provides codegenerator
                   requires generate_instruction, register_allocator
  realization ..(creates CODE603)
end codegen

subsystem generate provides generate_instruction
                    external symtbl
  realization .. (creates GEN07)
end generate


subsystem allocate provides register_allocator
  realization ..(creates REGALLOC)
end allocate.

realization .. (creates COMP01)

end compile
```

Figure 8. Partial Cooprider's MIL (Continued)

Figure 9. System and module families in Intercol.



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. The *scan* module in the INTERCOL version of our compiler example (Figure 10) shows the case of three different implementation modules using the same interface description.

INTERCOL interacts with a number of different programming languages by means of a *resource-specification sublanguage*. Resources are constructs in a specific programming language that are implemented and used in the modules. A mapping from a 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 Ada.

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.) as illustrated by module symtbl of our example (Figure 10) and a specification is a list of resources written in the resource specification sublanguage.

A module family has an interface consisting of a list of **provided** and **required** resources and contains one or more implementations. Each implementation may exist in several revisions that are the entities or files that contain the actual programs. Different programming languages can be used for different realizations. Each realization may have several revisions, where a revision is the result of programming the initial revision or editing an existing one. Derived versions constitute a second dimension of variations of realizations.

A system family contains zero or more module and system families and zero or more compositions. A composition gives a name to a combination of elements that are the names of previously declared building blocks in the same or enclosing system families. The *compile* system of Figure 10, for example, could have other compositions in addition to the one indicated for PL/1 if implementations of the other modules (e.g., Pascal) were available. If one compares the INTERCOL version of our example with the other MIL descriptions, it will be noticed that there is an increase in the detail of the interface description. It could be said that INTERCOL is "stronger typed" than the other MILs. The increase in detail is needed for more effective type checking but it forces the system developer into premature decisions about module implementation.

An INTERCOL user and a Cooprider's MIL user would follow an almost identical process in constructing a software system. INTERCOL, however, is imbedded in a Software Development Control Facility (SDCF) which is organized as an interactive system that controls a software development database. The SDCF, moreover, allows for independent and incremental compilation and type checking, thus, significantly reducing development costs.

The advantage of using Tichy's SDCF rather than the previous MILs occurs in controlling the evolutionary process of a software system. The approach of system design by "evolving prototypes" would be the ideal approach to use with this SDCF.

Differences from the other MILs. The most significant contributions of INTERCOL and Tichy's SDCF to MILs are that they

1. Allow a structured specification and control of families of systems and modules
2. Allow independent compilation and type checking
3. Include an interface control system that automatically manages the consistency of the interconnection among module and system families
4. Include a version control system that supervises the addition of new versions.

Experience to date. Tichy's SDCF became operational at the prototype level in a PDP 11/40 system under UNIX and was later integrated into the Gandalf System [24] as the System Version Description facility.

3.5 Observations

We use the compiler example as the basis for our observations. MIL75 introduces the basic MIL constructs—requires, provides, has-access-to, etc. With these basic constructs expressed in a formal syntax, a system structure can be completely described and automatically checked for inconsistencies. Our compiler example was overdecomposed to emphasize the interconnection structure. An MIL75 description is as rigid as its ancestor—a structured design chart. The construction of an MIL75 structured description usually follows a preordered traversal of the structured design tree. The language, although rigid, is effective in enforcing a proper structure.

Thomas' MIL simplifies the notation, and provides a more flexible approach to describe system structure. Nodes are not required to be listed in a rigid nested tree structure. Thomas' MIL provides the syntactic construct **using** to indicate the particular program that implements a given module. It can be considered as a rudimentary approach to version control. By providing more flexibility in module arrangement, Thomas' MIL makes information hiding easier to implement as shown in the scan node (Figure 7).

```
System compile
   provide compile

   module compiler /*compiler driver-main program*/
      provide procedure compiler
      require scanner, parser, postfix_generator, symtbl_funct
   end compiler

   module scan
      provide package scanner is
         type token:ALPHA
         function nextchar_funct return:CHARACTER
         procedure backup_proc
      end scanner
      require symtbl_proc, errormsg_proc
      implementation SCAN02 .. PL/1
      implementation SCAN03 .. NIP
      implementation SCAN04 .. Pascal

   end scan

   module parse
      provide procedure parser
      require scanner, postfix_generator, system-label_proc,
               errormsg_proc, verify_funct
      implementation PARSE.NIP
   end parse

   module postfix
      provide package postfix_generator (token) is
         procedure codegenerator
         procedure generate-instruction
         function register-allocate return:INTEGER
      end postfix_generator
      require system-label_proc, errormsg_proc
      implementation POSTFIXGEN.NIP
   end postfix

   module symtbl
      provide package symtbl_proc
         constant tblsize:INTEGER, found:LOGICAL
         type name, type, procname:ALPHA
         type location is range 1..tblsize
         procedure enter-proc (name, type, procname, location)
         procedure get-proc (name, type, procname, location)
         procedure lookup-proc (name, type, procname,) return
                                          location found;
      end symtbl_proc

      require errormsg_proc
      implementation SYMBL-PL/1
      implementaiton SYMBL-Pasc
   end symtbl

   module system-label
      provide procedure system-label_proc (label:name)
                                             /*create new label*/
      require symtbl_proc
      implementation SYSLBL
   end system-label

   module verify
      provide function verify_func (token) return:LOGICAL
      require errormsg_proc
      implementation VRFY
   end verify

   module errormsg
      provide procedure errormsg_proc
      implementation ERRMSG
   end errormsg

   COMPOSITION
   compile_A = [compile, SCAN02, PARSE.NIP, SYMBL.PL/1, POSTFIXGEN.NIP
                        SYSLBL, VRFY, ERRMSG]
end compile
```

Figure 10. INTERCOL code for a single one pass compiler.

Cooprider's MIL provides a complete version control scheme and a specialized construction syntax to describe how a software system is constructed by assembling existing modules selected from a variety of different versions. The interconnection syntax is simplified even further to only three constructs: **provides, requires,** and **external.** Multiple nesting is allowed, thus information hiding can be implemented at different levels of abstraction.

Intercol expands the version control scheme to include families of systems and modules. It provides for a module description to be the standard description for a family of modules at the expense of some explicit detail in the resource types. Given an interface description, modules may be implemented in different languages or for different target machines.

In summary, it can be observed that MILs follow an evolution development from basic—simple MIL75—to the more powerful and elaborate Intercol. They all share the same interconnections constructs but differ in their implementation and on the extra features to aid the software development process.

In the next section, several software development systems are surveyed and classified by the type and level of module interconnection they support.

SYSTEMS SUPPORTING MODULE INTERCONNECTION

There are several programming environments and software development systems that provide module interconnection facilities. Detailed classification of these systems is somewhat difficult since each has its own unique characteristics and has emerged from a distinct design philosophy. From the point of view of MILs, however, we have selected a representative sample of the different approaches taken to do module interconnection. The sample includes the following systems: PWB, CLU, ADAPT, MESA, PROTEL, CDL2, SARA and Gandalf. Each of these approaches is discussed in more detail in the following sections.

PWB [27] represents the class of systems that provide facilities for management of systems development (i.e., version control) but lacks facilities for strict module interconnection (i.e., intramodule typechecking, systems structure description, module accessibility). Systems like the Software Factory [8], the SWB System [31], and the ARCTURUS System [46] also fall into this class.

CLU and ADAPT represent the class of languages and language extensions perfectly suited to support module interconnection. Languages in this class are highly modular and provide constructs for version definitions. They are based on data abstractions and use the same language for module construction (PS) as for system description (PL). This last characteristic may be considered a drawback from the point of view of MILs. MODULA, ALPHARD, and Ada also fall into this class. Within this category we include Prolog [26] as an effective MIL with built-in powerful automatic processing and the additional feature of automatic evaluation of design metrics. An awkward notation is one of Prolog's disadvantages.

MESA, PROTEL, and, CDL2 represent the class of fully integrated software development systems that is actually used in production environments. This class of systems performs module interconnection as MILs do but are restricted to modules written in their own language.

SARA represents the class of special purpose software design systems at the development stage that uses a different approach to module interconnection—SARA advocates a formal model approach instead of MILs. A MIL based on Ada, however, has recently been added to SARA [5]. This Ada based MIL is one of the best examples of the potential of Ada as both an LPL and LPS.

Gandalf represents the class of fully integrated software development systems that has successfully included an MIL—INTERCOL—as one of their development tools.

4.1 PWB

The PWB (*P*rogrammer's *W*ork *B*ench) facility provides limited support for module interconnection. Based on UNIX, PWB was developed by Bell Labs in 1973 [6, 15, 27] 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 are different than those required by the program users.

PWB succeeds in separating the program development and maintenance function onto a specialized computer that is dedicated to that purpose. This computer provides the interface between program developers and their target computer(s). PWB supplies a separate uniform environment in which people perform their work. The facilities supported by the PWB 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 *S*ource *C*ode *C*ontrol *S*ystem (SCCS) [43] is a file storage system that records the vari-

versions of a text file; this is accomplished by recording the original version plus interleaved modification descriptions that can be applied to create more up-to-date versions. This system supports creation of any revision of a source program or text, file protection against accidental changes, selection propagation of module changes to each of its revisions, and identification of object and source (revision number, date created, etc.). PWB's SCCS System does not provide, however, syntax constructs for module interconnection descriptions as an MIL would do.

4.2 CLU

The programming language CLU was designed by Liskov [30] to implement the concept of abstract data types. It provides constructs that support the use of abstractions in program design and implementation. An earlier language, ALPHARD [51], was designed mainly to support the construction of structured programs. Both, CLU and ALPHARD deal with abstract data types and abstraction building mechanisms and both are essentially derived from SIMULA 67 [13, 7]. Although CLU and ALPHARD are somewhat similar, they differ in many important details.

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

For each abstraction there is a *description unit* which contains all system-maintained information about that abstraction. The *interface specification* which is that information needed to type-check uses of the abstraction is the most important information about 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 constraints 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 a module's external references 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, and this list 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 same idea as MIL75. An important feature of CLU is its type checking capability across modules, which is a natural consequence of its objective: to aid the programmer to construct correct programs. A drawback is its lack of support for system organization.

Cooprider showed that an MIL based on a database processor is more effective in the control of system organization than an MIL based on a compiler. It could be argued that the CLU library is the equivalent of a database processor because it supports incremental program development, but cannot support version nor system family control because the compiler binds a module permanently to the abstraction it uses. This is the price of the strong type checking needed for correct programs. CLU therefore is more of an LPS *L*anguage for *P*rogramming in the *S*mall than an LPL.

4.3 ADAPT

*A*bstract *D*esign *A*nd *P*rogramming *T*ranslator, a language resembling CLU in its essentials but with PL/1-style syntax, has been implemented at IBM [1, 2]. It has proven to be as good a mechanism for describing the detailed semantics of modules as CLU, but in contrast to CLU, an MIL has been added. This MIL extension to ADAPT is called *External Structure*.

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

The ADAPT language is used to define the semantics of modules. There are three kinds of semantics, corresponding to procedural, data, and control abstractions and defined by the PROCEDURE, CAPSULE, and ITERATOR constructs respectively.

PROCEDURE is like procedures in other languages, but parameters may be used-defined data types, in addition to the data types provided by the language. CAPSULE is used to specify a data abstraction and consists of an internal data representation and operations that manipulate the internal representation. ITERATOR defines control abstractions by defining

how elements of an abstract data collection are obtained so that actions on the elements can be programmed independently. Iterations may be encapsulated within a data abstraction, or they may exist as an abstraction in their own right.

The External Structue language is used to describe systems as collections of modules and their interconnections. The definition of a module includes the name of each interface, the types of the parameters, and the return value for the interface.

Procedures are not required to have a return value, and their functional type specification has abbreviated syntax. For example, a WRITE procedure to transmit a STRING to a printer has the following syntax:

WRITE(STRING)

When the module is a data type, the interface is described by a set of operators; i.e., by encapsulated procedures and iterators called the DEFINES list for the module.

In addition to the interface definitions, the modules used by the module being described are listed in a USING list. Although USING lists are required for compilation, they can be deferred or abbreviated allowing the designer to defer representation decisions during the early stages of design.

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

4.4 MESA

MESA was developed at Xerox PARC during 1975 [22, 33] 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 an operating system is reported in [29, 25].

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

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

From MIL's point of view, MESA and C/MESA form a well-integrated set of tools covering the design and implementation aspect of a software system lifecycle. The MESA System succeeds in implementing some of the ideas originated in MIL75 and parallels some of the ideas of Cooprider and Tichy on version control but at a less general level. The goal of C/MESA is to allow the user to represent a complete system in a hierarchy of configuration descriptions. In MIL75 terms, C/MESA has all the syntactic constructs to represent a system tree.

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

A module declaration in MESA is a list of variable names and the names 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 language for PS. 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 the *implementor* of these facilities.

An interface consists of a sequence of declarations defined by a *definitions* module. Only the names and types of operations are specified in the interface, not their implementations. A definitions module and one of its implementors is illustrated in Figure 11 (taken from [22]). Modules and interfaces are compiled separately. The compiler reads each of the imported modules and obtains all the information necessary to compile the importing module, performing type-checking for all references. No knowledge about any implementors of the interfaces is required.

The MESA binder collects exported interface records that have been identified with a unique name by

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

Implementor: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 11. Definitions module and an implementor in MESA.

the compiler, and assigns their values to their importer's corresponding interface records. 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 code to bind modules together to form configurations. The partial code for a system configuration is shown in Figure 12 (taken from [22]). In this example, **A, B, C,** . . . are the interfaces and **U, V, W,** . . . are the modules that import/export them as indicated by the special comment characters–.

The definition modules in MESA are equivalent to the declarative statements of any of the MILs described above, and the separate C/MESA code is equivalent to an 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 11. The same could be said if Ada were used as an MIL.

Figure 12. A partial configuration description in C/MESA.

```
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.
```

The separate C/MESA code, as illustrated by the example of Figure 12, 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.

4.5 PROTEL

*PR*ocedure *O*riented *T*ype *E*nforcing *L*anguage is a tool that supports type checking across modules in a fashion similar to MESA [11]. 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 in-

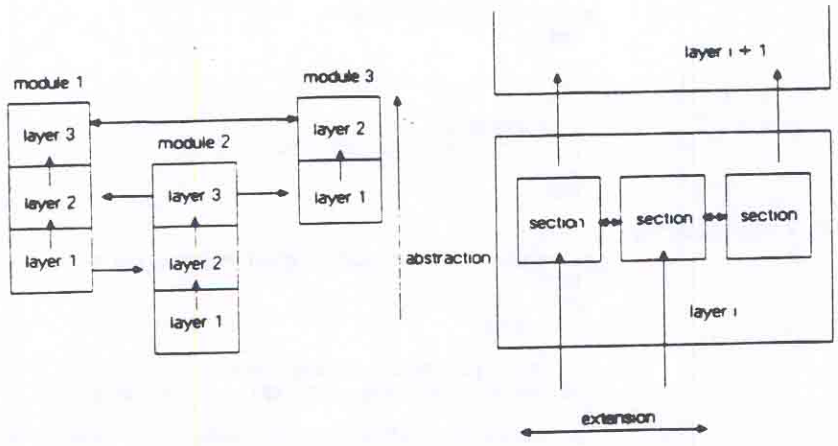


Figure 13. Module structure in CDL2.

tersection and intermodule references, the compiler performs a process called *embedding* which consists of first writing symbolic information to an object file, reading that information for all sections visible to the one being compiled, and using that information 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 [11]. PROTEL is very limited in controlling system versions and in supporting system organization. The Library System is restricted to modules coded in PROTEL.

4.6 CDL2

The CDL2 system, developed at the Technical University of Berlin between 1977 and 1980, is a development environment centered around a single language [4]. It is intended for the development of large, sequential systems.

From the MILs' point of view, a CDL2 program consists of *modules*, which may be connected via explicit *export* and *import* interfaces. Each module contains a strict hierarchy of *layers*. Lower layers can export resources to layers at a higher level of abstraction. In Figure 13(a) horizontal arrows represent import/export module interfaces and upper pointing arrows show layer export direction.

Within the layers is found a set of *sections* connected by explicit interfaces to other sections in the same layer or in the next higher. Export of resources from one section to another within the layer is called *extension;* exports to sections in the next higher layer are called *abstractions*. Sections are functional units like modules. Extension is used to extend the power of a layer; abstraction is used to realize one level of the abstract machine in terms of the next lower one. The division of modules into layers and of layers into sections allows the user to define different layers of interconnection. If a module, for example, is originally designed at a very low level of abstraction, a higher level description of the same module could be designed to provide a simpler, more general interconnection interface.

The unique hierarchical organization of modules in CDL2 provides the structure for describing a very high-level design that could accommodate different versions of the same module. The lower level layers of a particular module could be replaced by layers performing the same function but having different characteristics. This is analogous to the family and version control of Cooprider's MIL and Tichy's INTERCOL.

The CDL2 System is centered around a command interpreter that provides the user with a uniform language to control all components of the system (Editor, Formater, Analyzers, Coders, and Database). From the MILs point of view, the Local and Global Analyzers are essential because together they perform the role of an MIL processor (i.e., compiler). The *Local Analyzer* consists of a *Syntax Checker* and a *Local Semantic Checker*. The *Global Analyzer* consists of a *Global Semantic Checker* and an *Intermodule Interface Checker*. The Intermodule Interface Checker is used during system design and specification to create a general design description. The Global Semantic Checker verifies in

port/export data types across modules (similar to the MIL75 compiler). The Local Semantic Checker verifies internal module interfaces (among layers).

The CDL2 System is presently being used in various research projects within the Technical University of Berlin and has been transported to other sites where it is being used as an experimental software development environment.

4.7 SARA

*S*ystem *AR*chitect's *A*pprentice (SARA) is a computer-aided system that supports a structured multilevel requirement driven methodology for the design of reliable software or hardware digital systems. SARA was designed at UCLA in 1976 [18, 9, 10] 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 execution record, 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 (*G*raph *M*odel of *B*ehavior).

The structural model resembles the *contour model* [28] used to describe the semantics of the execution record 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 the realization 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 similar to 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.

In mapping between the behavioral and structural models, SARA structures the execution record. 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.

In 1979, an MIL was added to SARA [38] to deal with algorithm structure. This MID (*M*odule *I*nterface *D*escription) was intended to enhance SARA's power by providing a smoother path from modeling to code.[2]

In this new model, a SARA-MID 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-MID) says which variable is of which type and which call is of which procedure.

More recently, Berry [5] proposed on Ada based MIL tool for the SARA System. This Ada based MIL satisfies the basic MIL requirements of module description and interface definitions including the equivalent of **provide** and **uses** constructs.

Berry uses a subset of Ada for this purpose which allows as a major advantage that a normal Ada compiler can perform module interconnection consistency checking. The MIL subset of Ada comprises the object, type, subprogram, package, exceptions, and renaming declarations and the generic versions. It also includes procedure and function calls, and compilation units.

A MIL description of a system consists of a collection of package and procedure compilation units, one for each module. The specification parts of these units make explicit what resources they provide.

Berry introduces a generic procedure that allows the user to invoke provided resources that are usually value returning functions. This generic procedure is instantiated with a null body for each type that appears in the MIL descriptions. The purpose of this procedure is to provide a "receptacle" for the returned value no matter what type it is. The MIL programmer may invoke this procedure around any invocation of a function that is written in the specification. This way the Ada compiler does not complain about the return value not being used and does all the desired parameter checking.

The proposed Ada subset does not include tasking. Berry observes that "the differences between an abstraction built with tasks and one functionally identical built without tasks are strictly of behavior and performance." Thus tasking is not part of the code structure language but belongs to SARA's other languages.

Examples coded in this Ada based MIL, including Parnas' KWIC system [37], have been tested successfully with the NYU Ada/Ed compiler. The compiler does all interconnection verification except for warning when a module (an abstraction in Ada) is not used by any other module and if none of its provided resources are involved.

---

[2]Formerly called MISC for *M*odule *I*nterconnection *S*pecification *C*apability.

The potential for integration of this MIL in Ada environments is one of the major advantages of this approach. A disadvantage, at least from the point of view of this survey, still remains—a single programming language. Another disadvantage, from the systems designer perspective, may be the high level of detail required to describe a module interface. All types must be defined at module description time.

4.8 Gandalf

Gandalf [24, 21] is a new software development environment, different from all the conventional tools, such as the ones described above. It is designed for projects that use Ada, but its current implementation is written in the C language.

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

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

The incremental program description tool consists of a *syntax directed editor*, ALOE [32] and a *syntax directed dynamic debugger*, LOIPE [19]. 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.

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

The System Version Description and Generation Tool consists of both Cooprider's version control system and Tichy's SDCF. They address the two basic problems of system composition: module interface control and system version control. It provides a system generation facility based on system descriptions thus taking over all necessary bookkeeping from programmers or system builders, a 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 SDC is also responsible for avoiding conflicts of interest among project programmers; i.e., it will not permit two programmers to alter source code concurrently. Access rights are automatically checked by the system so that unauthorized users may not manipulate the product.

Documentation control 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 all changes made to the system state are 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 environments of the 1980s. It is built on most of the ideas described in the previous sections. it uses the concept of structured programming and stepwise refinement for construction of modular programs, Parnas' ideas [37] for module construction using information hiding, the concept of separating system specification (LPL) from implementation (LPS), system version representation by abstract data types, and several other ideas from previous tools, (e.g., UNIX, MESA, CLU, etc.).

An environment that could be considered similar to Gandalf is the Adele research project of the Laboratoire de Genie Informatique, IMAG, Grenoble, France [16, 17]. This project has four main components: (1) a program editor, interpreter, and debugger; (2) a parametrized code generator; (3) a user interface; (4) a program base. Components (1) and (2) are analogous to the incremental program construction tools in Gandalf and the program base is essentially a system composition and version control tool. Project management tools in the Adele project are presently limited to version numbering and version status (revisions) with explicit support for documentation generation and control to be added in the near future.

The program base is essentially a DB based MIL

similar to Tichy's. The version control mechanism is also based on Tichy's scheme but with an additional feature. Propagation of version changes and revisions are conducted automatically. The main original points in the Adele approach lie in the "expression of multi-version system composition by constraints of attributes rather than by component names (the attributes being locally attached to any component or configuration), and in the extended notion of 'status'." Estublier believes that such an implicit definition is often a more natural specification means for the designer than the explicit naming of components. This approach, however, departs from the idea of explicit descriptions advocated in MILs.

The current version of the program base is written in Pascal to support Pascal programs and runs in the Multics environment. It has been used in several experimental program developments with satisfactory results. The Adele project objective is to create an integrated software development environment.

Many new software development environments are essentially special purpose tools integrated in a system with an underlying database system. Most of them consist of the three basic development tools: program construction, system description/design and version control, and project management. The ECLIPSE system, for example [45], one of the newest of these environments, has an underlying structure very similar to Gandalf and Adele. A commonality of this kind of environments is the central role of the system description and version control tool which is basically an MIL.

4.9 Tool Support for Interconnection

The relationship among the tools described above and their support of module interconnection are illustrated below.

Full MIL Support: Gandalf, C/MESA, SARA-MID, CDL-2, Intercol, Cooprider's MIL, Thomas' MIL, MIL75

Partial MIL Support: Protel, Ada, MESA, PWB, SARA

Marginal MIL Support: CLU, SIMULA, MODEL, ALPHARD, modular languages

5. SUMMARY

After taking the reader through this detailed description of module interconnection languages and of some software development systems that support module interconnection, it is appropriate to mention their main contributions to software engineering technology.

MILs and their related processors represent a set of tools that 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, an MIL must be integrated into a software development system or facility where the MIL description of the system is checked every time a change is made to that system.

MILs' 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.
3. MILs can represent a system construction process and serve as the basis for a unified database during system development.

A consequence of these contributions is a substantial improvement of the maintenance stage. A designer can revise, modify, and type check a system at the MIL level before attempting any changes to the code.

These contributions, although significant, are only a modest advance towards improving software development technology.

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 ideas to drive research in other areas 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 to encode general specifications and how we can match requirement specifications with provision specifications. Among the issues to be addressed with this proposition are compatibility, functional equivalence, and uniformity. Reusability is also an important issue directly related to this matching scheme.

Reusability, as proposed by Freeman [20], should seldom deal with executable code, but should primarily use nonexecutable work products from system analysis and design. A research question is then how could an MIL be expanded or augmented to include information about the 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 [40] and NOPAL [44] are two nonprocedural languages used for automatic generation of computer programs that support module description and provide limited module interconnection. There is, however, a need for 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 to 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 nonexecutable work product). To reuse a construction process would be more attractive than reusing a design specification.

An inherent property of large systems is massive change over a long period of time. These changes occur in three ways: 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 and Tichy started to examine the problem of version control but more work is needed. The problem of which changes are inherited by a system that is new along some dimension 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 determined out that there is still a long way to go a before a major breakthrough in the manufacture of software is achieved. Every major breakthrough in technology, however, has been attained through small steps.

ACKNOWLEDGMENTS

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

REFERENCES

1. J. L. Archibald, The External Structure: Experience with an Automated Module Interconnection Language, *The Journal of Systems and Software 2*, 147–157 (1981).
2. J. L. Archibald, B. M. Leavenworth, and L. R. Power, Abstract Design and Program Translator: New Tools for Software Design, *The Journal of Systems and Software* 170–187 (1983). Also in *IBM Systems Journal*, 22, 170–187 (1983).
3. R. M. Balzer, N. M. Goldman, and D. Wile, On the Transformational Implementation Approach to Programming, *Proceedings; 2nd International Conference on Software Engineering* 337–344 (1976).
4. M. Bayer et al., Software Development in the CDL2 Laboratory, *Software Engineering Environments*, North-Holland, New York, 1981, pp. 97–118.
5. D. M. Berry, On the Use of Ada as a Module Interconnection Language, *Proceedings; 17th Annual Hawaii International Conference on System Sciences* 294–302 (1984).
6. M. H. Bianchi and J. L. Wood, A User's Viewpoint on the Programmer's Workbench, *Proceedings; 2nd International Conference on Software Engineering* 198–199 (October 1976).
7. G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard, *Simula BEGIN*, Petrocelli/Charter, 1973.
8. H. Bratman and T. Court, The Software Factory, *IEEE Computer* 28–37 (May 1975).
9. I. M. Campos and G. Estrin, SARA Aided Design of Software for Concurrent Systems, *Proceedings; National Computer Conference*, 1978.
10. I. M. Campos and G. Estrin, Concurrent Software System Design Supported by SARA at the Age of One, *Proceedings; 3rd International Conference on Software Engineering* 230–242 (May 1978).
11. P. M. Cashin, M. L. Joliat, R. F. Kamel, and D. M. Lasker, Experience with a Modular Typed Language: PROTEL, *Proceedings; 5th International Conference on Software Engineering* 136–143 (March 1981).
12. L. W. Cooprider, *The Representation of Families of Software Systems*, Ph.D. thesis, Carnegie-Mellon University, Computer Science Department, April 1979 (CMU-CS-79-116).

13. O. J. Dahl, B. Myrhaug, and K. Nygaard, The SIMULA 67 Common Base Language, Norwegian Computer Center Technical Report S-22, 1970.
14. F. DeRemer and H. Kron, Programming-in-the-Large versus Programming-in-the-Small, *IEEE Transactions on Software Engineering*, June 1976, pp. 321–327. This paper was presented at the International Conference on Reliable Software, Los Angeles, California, April 1975.
15. T. A. Dolotta and J. R. Mashey, An Introduction of the Programmer's Workbench, *Proceedings; 2nd International Conference on Software Engineering*, pp. 164–168, October 1976.
16. J. Estublier, S. Ghoul, and S. Krakowiak, Preliminary Experience with a Configuration Control System for Modular Programs, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 23–25, April 1984, Pittsburgh, PA.
17. J. Estublier, S. Krakowiak, J. Mossiere, and Y. Rouzaud, Design Principles of the Adele Programming Environment, *Proceedings of the International Computing Symposium on Application Systems Development* ACM, Nuremberg, March 1983.
18. G. Estrin, A Methodology for Design of Digital Systems—Supported by SARA at the Age of One, *Proceedings; National Computer Conference*, 1978.
19. P. H. Feiler, *A Language-Oriented Interactive Programming Environment Based on Compilation Technology*, Ph.D. dissertation, Carnegie-Mellon University, Computer Science Department, May 1982.
20. P. Freeman, Reusable Software Engineering: Concepts and Research Directions, in *Workshop on Reusability in Programming*, Alan Perlis, ed., pp. 2–16, ITT Programming, Newport RI, September 1983.
21. The Gandalf Project, *Journal of Systems and Software* 5, 2 (May 1985).
22. C. M. Geschke, J. H. Morris, and E. H. Satterthwaite, Early Experience with MESA, *Communications of the ACM* 540–552 (August 1977).
23. D. Gries, *The Science of Programming*, Springer Verlag, New York, 1981.
24. N. Haberman, D. Perry, P. Feiler, R. Medina-Mora, D. Notkin, G. Kaiser, and B. Denny, *A Compendium of Gandalf Documentation*, Carnegie-Mellon University, Pittsburgh, PA, 1981.
25. T. R. Horsley and W. C. Lynch, Pilot: A Software Engineering Case Study, *Proceedings; 4th International Conference on Software Engineering*, pp. 94–99, Munich, Germany, September 1979.
26. D. C. Ince, Module Interconnection Languages and Prolog, *SIGPLAN Notices*, pp. 89–93, August 1984.
27. E. L. Ivie, The Programmer's Workbench—A Machine for Software Development, *Communications of the ACM* 749–753 (October 1977).
28. J. Johnston, The Contour Model of Block Structured Processes, *SIGPLAN Notices—Proceedings Symposium Data Structures and Programming Languages* 55–82 (1971).
29. H. C. Lauer and E. H. Satterthwaite, The Impact of MESA on System Design, *Proceedings; 4th International Conference on Software Engineering*, pp. 174–182, Munich, Germany, September 1979.
30. B. Liskov, A. Snyder, R. Atkinson, and C. Shaffrt, Abstraction Mechanisms in CLU, *Communications of the ACM* 564–574 (August 1977).
31. Y. Matsumoto, et al., SWB System: A Software Factory, *Software Engineering Environments*, North-Holland, 1981, pp. 305–318.
32. R. Medina-Mora, *Syntax-Directed Editing: Towards Integrated Programming Environments*, Ph.D. dissertation, Carnegie-Mellon University, 1982.
33. J. G. Mitchell, W. Maybury, and R. E. Sweet, *Mesa Language Manual*, Tech. Report CSL-79-3, Xerox Corporation, Palo Alto Research Center, April 1979.
34. J. M. Neighbors, *Software Construction Using Components*, Ph.D. thesis, University of California, Irvine, 1980.
35. J. M. Neighbors, The Draco Approach to Constructing Software from Reusable Components, *IEEE Transactions on Software Engineering*, pp. 564–573, September 1984.
36. M. Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press, 1980.
37. D. L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM* 1053–1058 (December 1972).
38. M. H. Penedo and D. M. Berry, The Use of a Module Interconnection Language in the SARA System Design Methodology, *Proceedings: 4th International Conference on Software Engineering*, pp. 294–307, 1979.
39. R. Prieto-Diaz and J. M. Neighbors, *Module Interconnection Languages: A Survey*, Tech. Report UCI-ICS-TR189, University of California, 1982.
40. N. S. Prywes, Automatic Generation of Computer Programs, *Advances in Computers*, Academic Press, New York, 1977.
41. C. Rich, H. E. Schrobe, and R. C. Waters, Overview of the Programmer's Apprentice, *Proceedings; 6th Joint Conference on Artificial Intelligence*, pp. 827–828, 1979.
42. C. Rich and H. E. Schrobe, Initial Report on a Lisp Programmer's Apprentice, in *Interactive Programming Environments*, pp. 443–463, (D. R. Barstow, H. E. Schrobe, and E. Sandewall, eds.), McGraw-Hill, New York, 1984, pp. 443–463.
43. M. J. Rochkind, The Source Code Control System, *IEEE Transactions on Software Engineering* 364–370 (December 1975).
44. R. Sangal, *Modularity in Non-Procedural Languages Through Abstract Data Types*, Ph.D. thesis, The Moore School of Electrical Engineering, University of Pennsylvania, August 1980.
45. I. Sommerville, and R. Thomson, The ECLIPSE System Structure Language, *Proceedings; 19th Annual Hawaii International Conference on System Sciences*, pp. 413–419, 1986.
46. T. A. Standish, ARCTURUS An Advanced Highly-In-

tegrated Programming Environment, *Software Engineering Environments*, North-Holland, 1981, pp. 49–60.
47. W. P. Stevens, G. J. Meyers, and L. L. Constantine, Structured Design, *IBM Systems Journal* (1974).
48. J. W. Thomas, *Module Interconnection in Programming Systems Supporting Abstraction*, Ph.D. thesis, Brown University, June 1976.
49. W. F. Tichy, Software Development Control Based on Module Interconnection, *Proceedings; 4th International Conference on Software Engineering*, pp. 29–41, September 1979.
50. W. F. Tichy, *Software Development Control Based on Systems Structure Description*, Ph.D. thesis, Carnegie-Mellon University, Computer Science Department, January 1980.
51. W. A. Wulf, *ALPHARD: Toward a Language to Support Structured Programs*, Tech. Report, Carnegie-Mellon University, Computer Science Department, April 1974.
52. E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.