MODULE INTERCONNECTION LANGUAGES[1]

Dr. James M. Neighbors
November 1987

Module Interconnection Languages (MILs) are based on the difference between Programming-in-the-large (PL) and Programming-in-the-small (PS). The primary difference between PL and PS is that "structuring a large collection of modules to form a system (PL) is an essentially different intellectual activity from that of constructing the individual modules (PS)"[DERE76]. Architects of a large system are primarily concerned with the process of "knitting" system modules together rather than with the process of programming each module.

PS is concerned with building programs, with the particular use of loop-constructs, if-statements, assignment-statements, expressions, arrays, and so on. Over 35 years PS has been greatly developed to include the techniques of structured programming, typed-structure definition, top-down design, stepwise refinement, and others. The system lifecycle phases of *detailed design* and *implementation* primarily use PS notations. These notations focus on how a particular part (module) of a system performs its function.

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

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

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

1. copyright (c) 1987, 2001 Bayfront Technologies, Inc. Article written for Meridian Software Systems, Inc., newsletter. Meridian was an early provider of ADA compilers for the IBM PC.

1 The MIL Description of Systems

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

1. A PS (programming language) description of each of the modules in the system.

2. A PL (MIL resource language) description stating the resources provided and required by each module in the system.

3. A PL (MIL interconnection language) description of the resource flow between the constituent modules of the system.

In a MIL description, resources are any entity that can be named in a PS programming language (e.g. variables, constants, procedures, type definitions, etc.) and which can actually be made available for reference by another module within a given software system.

All resources are ultimately provided by modules, thus modules are units that *provide* resources and that *require* some set of resources. The primitive operations of a MIL describe the flow of resources among modules; they are **provide** (which may also be called synthesize or export) and **require** (which may also be called inherit or import). **Has-access-to** is primitive operation which checks to see if the named resource is visible at the given point of compilation. A **must** attribute may also precede the above operators.

The MIL description of a module specifies the resources required and provided by the module. This module description becomes the interface with other modules and subsystems and is made up of resource names and the operations which act upon them. Module descriptions are the actual code of a MIL and are used when assembling or integrating a software system in order to verify system resource flow.

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

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

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

        function XA
            must-provide a
```

```
          requires x
          has-access-to  module Z
          real x, integer a
     end XA

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

2 What MILs Do and Don't Do

Module Interconnection Languages provide the following abilities:

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

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

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

4. Provide maintenance management. A system can be revised, modified and type checked at the MIL level before attempting any changes to the code. MILs can prohibit programmers from changing the system architectural design during evolution and maintenance without an explicit change in the architectural design as represented by the MIL.

5. Manage version control and system family. This is an advanced but necessary function in developing large systems. A generalization of the construction process can be represented by a MIL and organized around a unified data base.

Aside from the basic operations listed above, a MIL usually serves as a project management tool by encouraging structuring before starting to program the details and as a support tool for the design process by capturing overall program structure and being capable of verifying system integrity before design implementation begins. A MIL also provides a means of standardizing communication among members of a programming team and of standardizing documentation of system structure.

On the other hand, some of the main limitations of MILs can also be listed.

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

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

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

3 Ada and MILs

Obviously, Ada provides support for MIL concepts. If we view Ada packages as the module units, then the package specification is the PL (MIL resource language) information and the package body provides the PS (programming language) information. The language provides for these two definitions to be packaged either separately or combined.

Ada provides weaker support for the third part of a MIL system specification, namely PL (MIL interconection language) information. The primary agents of module interconnection in Ada are the **with** clause, library unit management, and unit elaboration control. Investigation with MILs has shown that it is important for parent modules in a system to have the *ability* to control the resources requested and passed among its offspring. In Ada this would mean that a package A which is the parent by **with** context clauses of packages B, C, and D *could* control the context clauses of packages B, C, and D used in constructing those packages. This control is very fine, perhaps limiting B to only using some of the resources provided by C while allowing D access to a different subset of the resources provided by C. In addition, package A *could* decide which of the resources it has access to from packages B, C, and D are made available to packages which use it. Obviously elaboration becomes much more difficult.

We might solve this problem by generating different partially compatible **generic** or **renames** versions of C for use in the construction of A, B and D; but this seems like an extreme solution. Further, the hierarchy of modules making up a large system are usually a graph rather than a tree and that would make these solutions practically impossible.

Two limitations of the Ada elaboration environment make using MIL system descriptions difficult, 1) **with** clauses provide an “all or nothing” importation of visible objects from the specified package; and 2) the Ada unit library does not enforce any hierarchy among the packages or combination of packages. The hierarchy is required to describe the limitations in resource flow. It is important to note that these problems are not problems with the existing Ada language. Ada’s support for PS (programming language) and PL (MIL resource language) are very strong. Only the visibility of a package during elaboration is changed by PL (MIL interconnection language) definitions and these could be managed as part of the support environment (APSE). A MIL specifically designed for Ada is discussed by Tichy**[TICH80]**.

4 Further Information

MILs are very effective tools for aiding the architectural design of large systems which will be heavily maintained. A system must be evaluated, analyzed, and designed first by means of current methods and techniques. Once a system design is determined, it may be coded in a MIL to be checked and verified for completeness and inconsistencies. The significant support of architectural design as seen from the Software Engineering perspective, is what makes MILs an important tool for the software development process.

Further information about module interconnection languages may be found in the following:

[PRIE86] Module Interconnection Languages,
Ruben Prieto-Diaz and James Neighbors,
Journal of Systems and Software, 6, 307-334, 1986.

[DERE76] Programming-in-the-Large versus Programming-in-the-Small,
F. DeRemer and H. Kron, *IEEE Transactions on Software Engineering*, 321-327, June 1976.

[TICH80] *Software Development Control Based on Systems Structure Description*, W.F. Tichy, Ph.D. Dissertation, Carnegie-Mellon University, Computer Science Department, January 1980.