A Theory Model Core for Module Interconnection Languages

Thomas R. Dean
dean@qucis.queensu.ca

David A. Lamb
dalamb@qucis.queensu.ca

October 1994
Technical Report
ISSN-0836-0235-94-370

Department of Computing and Information Science,
Queen's University
Kingston, Canada K7L 3N6

Document prepared 10/7/94

Abstract: The Theory–Model paradigm is a way of thinking about design methods. The theory of a method specifies categories of information and axioms about them. A particular design is an interpretation of the theory if it assigns meaning to all of the categories, and is a model if it satisfies all of the axioms.

Module Interconnection Languages (MILs) describe and enforce the modular structure of systems. Originally developed in the 1970's, research in the area has reemerged in several incarnations over the years. While there have been differences in each approach, there are many features common to all of the MILs. This paper attempts to provide a characterization of the core features using the Theory–Model paradigm. In doing so, we hope to evaluate the use of the Theory–Model approach for comparing more complex design notations.

1. Introduction

The principle of information hiding [Parnas 72a, 72b] is considered a cornerstone of modern software design; it is has been incorporated into notations intended to capture portions of the design of a software system. One class of such notations is Module Interconnection Languages (MILs), which document the modular structure of a software system and are sometimes accompanied by environments to enforce the specified structure. Originally developed in 1975, research on these languages has continued sporadically over the years.

The Theory–Model [Ryman 89] paradigm is part of a long-term effort to study, formalize, and compare aspects of many different design methods. MILs are much smaller and simpler than complete design methods, so comparing them via the Theory-Model paradigm is a reasonable step towards our long-term goals.

We use the Theory–Model paradigm to compare several MILs. Our comparison is based on the representation of the system structure they provide. We begin by specifying a core representation shared by all of the MILs. We then characterize each of the MILs by the extensions to that core.

In 1986, Prieto-Diaz and Neighbors [Prieto-Diaz & Neighbors 86] presented a survey of the then current state of the art in MILs. Since then, several other notations have emerged. Prieto-Diaz and Neighbors argue that research in MILs can be viewed from three viewpoints: software engineering, formal models and artificial intelligence. They phrase the problem addressed by MILs as:

> “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?"

They claim that the software engineering viewpoint is to find a notation that accurately captures the design of a system. MIL system descriptions can be mechanically checked for consistency and completeness before the actual coding is completed. Prieto-Diaz and Neighbors survey the literature from this point of view.

Prieto-Diaz and Neighbors identify two formal model viewpoints of MILs. The first view is as a model of system resource usage during execution. The applications of this model are determining the data loading of the system, and detecting communication deadlocks. The other view within the formal model viewpoint is as a consistency model of system construction. This model uses restrictions on versions or implementations of modules to establish a set of modules that implement the system.

The artificial intelligence point of view identified by Prieto-Diaz and Neighbors is one of automatic programming. A knowledge base containing constraint and implementation information is used to implement a high level specification using lower-level steps. They claim that the module interconnection problem is making sure that the lower-level steps used are compatible.

We phrase the problem in a slightly different manner. MILs are used to represent the structure of a system, including the separation into independent elements. We rephrase the problem as:

> “How can a language or environment be designed to document the structure of the system?”

Some questions that arise from this point of view are:

- Where is the boundary between programming-in-the-large and programming-in-the-small?
- What is the relationship between programming-in-the-large and system structure.
- What are the language features appropriate to programming-in-the-large?

DeRemer and Kron [DeRemer & Kron 76] argue that programming-in-the-large requires a language that provides appropriate abstraction, structure and style. By finding the model common to all of the MILs, we can supply some answers to these questions.

We have examined seven module interconnection languages: MIL 75 [DeRemer & Kron 76], Thomas’ MIL [Thomas 76], Cooprider’s MIL [Cooprider 79], INTERCOL [Tichy 80], NuMIL [Narayanaswamy & Scacchi 87a, 87b] and Ryman’s notation [Ryman 89]. We do not consider the modular constructs provided by languages such as Modula–2 [Wirth 85] or Turing [Holt *et al.* 88]; besides restricting the module interconnection to a single language, the features provided by these types of languages are a subset of those provided by the surveyed MILs.

Module interconnection languages are relevant to software design for two reasons. The first is that they illustrate that different levels of abstraction or views of a system represent fundamentally different problems. The second reason that MILs are relevant to software design is that they attempt to represent the structure of a software system.

Section 2 presents the Theory–Model paradigm and it's mechanics. Section 3 provides a survey of the concepts common to all of the MILs. Section 4 presents an overview of some of the ways in which the MILs differ. The core theory and the extensions to the core that represent additional concepts provided by one MIL are examined in section 5. Some comments and conclusions are provided in section 6.

2. The Theory–Model Paradigm

The approach used in this paper is the same as used by Ryman [Ryman 89]. A *theory* provides categories of concepts, and rules that elements of those categories must satisfy. An *interpretation* is a collection of facts that are elements of the categories. The interpretation is a *model* of the theory if the facts satisfy the rules. In this view, borrowed from mathematics, interpretation corresponds to implementation and model to verified implementation. In conventional use, model typically means abstraction, which the Theory-Model paradigm would call a theory.

There are three steps to developing a theory using this approach. The first is a precise description of the categories and the restrictions on the categories. This is the most challenging step, accounting for 80% or more of the intellectual effort.

The second step is to capture as much of the theory as is possible using the Entity Relationship (ER) approach [Chen 76]. The ER schema provides a means of encapsulating domain and cardinality constraints on relationships between elements of the categories. It also provides a way of organizing large quantities of facts. One group of decisions that must be made is which categories are the base categories and which can be derived from others. Similar decisions must be made for the relations between the entity sets. There are several notations that may be used to build ER schemas. We use the Object Model Notation defined by Rumbaugh *et al* [Rumbaugh *et al.* 91]. It provides not only the ER elements of the notation (Entity sets, relations, cardinality), but also representations for derived entities, derived relationships and constraints. Not all of the restrictions will be absorbed into the ER schema. Other restrictions are expressed as constraints on the classes and relationship provided by the ER schema.

The last step in developing a theory is to translate the ER schema and the additional constraints to some inference engine such as Prolog. The base categories and relations are templates for facts while derived categories and relations are represented as rules that use the facts. The domain and cardinality constraints of the ER schema are easily translated to predicates. The other constraints are also coded as predicates, using the ER schema as a template. The inference engine provides a means of mechanically verifying that the facts are a model for the theory. We have postponed this step since it is not required for the kind of comparison we do here; the translation would be similar to what we have done in previous formalizations [Lamb *et al.* 90].

In the case of module interconnection languages, the theory is the syntax and semantics of the system representation translated to an appropriate notation. The representation of a particular system provides the facts (an interpretation) that form the model for the theory if it is legal in the MIL.

3 General Concepts

Modular interconnection languages were first introduced by DeRemer and Kron [DeRemer & Kron 76]. They argue that there is a significant difference between programming individual modules, and designing systems comprised of modules. DeRemer and Kron label the two activities programming-in-the-small and programming-in-the-large.

The advantages of modularization of a design are accompanied by an added complexity in the connections between the modules. Because of the fundamental difference between programming-in-the-large and programming-in-the-small, separate languages should be used for each activity. DeRemer and Kron argue that programming-in-the-large requires a language that provides appropriate abstraction, structure and style. One advantage of a separate language is that several languages for programming-in-the-small may be used.

By introducing the concept of programming-in-the-large as a different activity using a different language, DeRemer and Kron laid much of the groundwork for future research. Most of the general concepts in this section originate with them.

3.1 Purpose

The primitives in a MIL are modules and resources. DeRemer and Kron defines a module as a segment of code in a language for programming-in-the-small. Modules share resources which are entities from the domain of programming-in-the-small. Some examples are constants, types, variables and procedures. A MIL description of a module specifies what resources are provided and required by that module. The description also lists the modules that comprise a system, and specifies the exchange of resources between them.

There are two purposes to specifying resource flow between modules. The first is to resolve the required resources of each module with provided resources of other modules. The second, and possibly more important, purpose is enforcing disconnectivity between modules. That is, specifying the resource flow between modules allows us to later verify that the only connections present in the code are those that are in the specification. Any details of the internal operation of the modules are not included in the specification; as suggested by DeRemer and Kron, they are in the domain of programming-in-the-small.

Since the exchange of resources between modules is specified in a separate language, a compiler of some form is needed to process the description. All of the surveyed approaches propose some method of automatically checking a system description. There are two approaches to incorporating MILs into the development process. The first is as a separate tool [Thomas 76]; such processors check a system description for consistency, and check the implementation to make sure that it conforms to the system description. The other approach is to build the MIL into the development environment [Cooprider 79], [Tichy 80], [Narayanaswamy & Scacchi 87a]. In this situation, some form of project data base is used to keep track of MIL information. The system description is checked for consistency when it is entered into the database, and when any changes are made to the system description. The development tools check the description to make sure that any information conforms to the MIL system description.

As mentioned by Prieto-Diaz and Neighbors, MILs are only part of the design environment. The system under construction must be analyzed and designed using standard techniques such as stepwise refinement [Wirth 71] and information hiding. The MIL is used to specify the interconnection in the design, possibly as the design is evolved. Other information such as the functional specification of the modules must also be specified using other tools.

3.2 Functions of MILs

DeRemer and Kron suggest that a MIL should function as a design tool, as a project management tool, as a communication medium, and as a documentation method.

As design tools, MILs promote structure and the decomposition of a system into modules before any code is written. They can be used to establish the overall structure of systems, and can be used when combining existing software to make new systems. Prieto-Diaz and Neighbors note that it is easier to decompose the system into modules and specify those modules using a MIL than to design and implement the system and later extract a MIL description. The MIL can provide feedback in the early stages of system development before any code has been written.

As a project management tool, a MIL can record the development of the system. DeRemer and Kron[DeRemer & Kron 76] and Thomas[Thomas 76] claim that the MIL structure reflects the top–down decomposition of a problem into successively smaller problems. MILs also assist in the design of testing strategies. DeRemer and Kron suggest that testing at the level of individual modules and testing the system as a whole is inadequate. By documenting the structure of the system, plans that test interconnections can be developed.

The specification acts as a communication medium between the designers and programmers and between members of the programming team. It provides a written description of the interface that is enforced by the compilation system. Any code or implementation of a module must conform to this interface.

Finally, the MIL description of the system provides a method of documenting the system interconnection in one place. Without a MIL, the structure and interconnection information is spread between code, construction commands and documentation. Maintaining consistency between all of these media can be difficult. The MIL collects this information in one place, and the system construction process ensures that the description and implementation are consistent. DeRemer and Kron suggest that a MIL should also be used for proving-in-the-large, i.e. proving that correct modules will work when combined into a larger system. It may be that a different approach is used than that used to prove individual modules (proving-in-the-small).

MILs are also useful for system maintenance and evolution. Prieto-Diaz and Neighbors suggest that the largest gain from the use of MILs occur not from the design phase, but during system integration, and for system evolution and maintenance. The MIL description represents up-to-date documentation of the system structure and the dependencies between modules. The documentation remains up to date because any modifications to the system must conform to the MIL. If a modification is serious enough that it invalidates all or part of the MIL, then the MIL must be modified to reflect the new system structure before the system can be reassembled. By restricting inter module connection to that specified in the MIL, it is easier to replace existing modules with confidence that the system will work as expected.

3.2 Properties of MILs

The properties of MILs vary. In this section we discuss the common properties of all of the MILs surveyed: external scope of resources, static typing, functional specification and loading.

Modular interconnection languages define the scope of resource names outside of the modules that provide them. The system description is used to determine which systems and modules have access to each of the provided resources. The scope of the resources within the modules are defined by the languages used to implement the modules.

In all of the MILs surveyed, the module interconnection specified in the MILs are static. That is, the resources used by a module during execution can be determined at compile time. Most MILs do not specify the functional nature of resources. DeRemer and Kron suggest that while the functional specification of the modules is important, it is outside the scope of MILs. MILs should be one part of an environment where they are used for specifying the connections between modules, and some other notation is used to specify the functions of the modules. NuMIL [Narayanaswamy & Scacchi 87a, 87b] uses the concept of an abstract interface which may be used to interpret the functionality of a resource. The interconnection component of Ryman's notation [Ryman 89] is embedded in English text which describes the resources in greater detail.

None of the MILs surveyed address the problems of program loading. Program loading problems include the division of the system into overlays, and the scheme used to load the system into memory during execution. Ryman [Ryman 89] supports the concept of drivers and automatically generates segmentation information for libraries, but the interconnection notation does not address these issues.

4. Major Ideas

In this section we review the current literature and compare the different approaches and features of each of the MILs. Not all of these features are relevant to the core model, but are necessary for an understanding of the issues that MILs address. Table 1 provides a summary of these ideas. The areas we discuss are:

	Mil 75	Thomas	Cooprider	INTERCOL	NuMIL	Ryman
Modules	Segments of LPS	Single Resources	Segments of LPS	Segments of LPS	Segments of LPS	Packages
Resources	LPS Constructs	Procedures & Clusters	Ascii String	LPS Constructs	Abstract Constructs	LPS Constructs
Approach	Custom Language	Custom Language	Custom Language	Custom Language	Custom Language	Logic Systems
System Representation	Tree	Directed Graph	Nested Blocks	Nested Blocks	Dir. Acyclic Graph	Set Theory
Resource Access	Graph	Graph w / Abstraction	Block Structure	Block Structure	Block Structure	General
Resource Types	None	Procedures & Clusters	None	Resource Language	Abstract Interface	Methods & Datatypes
Resource Translation	Not Discussed	Not Discussed	Processors	Built In	Single Language	Not Discussed
Environment Support	Not Discussed	Separate Tool	Project Database	Project Database	Project Database	Knowledge Base
Enforcement	Separate Tool	Separate Tool	Environ-ment	Environ-ment	Environ-ment	Separate Tool

Table 1 – MIL Major Ideas

- Approach.
- The primitives of the MIL.
- System definition.
- Resource access.
- Resource typing.
- Resource translation.
- Environment support.

4.1 Approach

The MILs surveyed can be broken down into two approaches. These are the custom language approach and the logic system approach.

The custom language approach uses a language designed specifically for MILs. System descriptions are used by dedicated tools to check the consistency of the design. The implementation can be checked against the design specification using the MIL software, or the development tools used for programming-in-the-small can be modified to check the implementation against the MIL system specification. The MILs that use this approach are MIL 75, Thomas' MIL, Cooprider, INTERCOL, and NuMIL.

In the logic system approach, the system design is represented and checked using first order logic. This approach is used by Ryman. In Ryman's case, some of the code for the system is automatically generated.

4.2 MIL Primitives

Not all of the MILs define the primitives in the same way. Differences in the interpretation of the primitives lead to differences in the MIL notations and in their use. All of the MILs have primitives corresponding to modules and resources.

Resources. DeRemer and Kron define resources as any nameable entity in the language for programming-in-the-small. INTERCOL and NuMIL also use this definition. Thomas restricts the resources to clusters (a form of Abstract Data Type) and procedures. Cooprider defines a resource as anything that can be represented as a character string. For programming languages, the string may contain external declarations of functions and variables. In documentation, an example of a resource might be a figure or diagram encoded in a formatting language. Ryman's parts are analogous to resources as defined by DeRemer and Kron.

Modules. DeRemer and Kron define modules as segments of code written in a language for programming-in-the-small that defines one or more named resources. This definition is shared by INTERCOL and NuMIL. Thomas' MIL is heavily influenced by Alphard [Wulf 74] and CLU [Liskov *et al.* 81]. He defines modules as segments of code that define a single resource. Cooprider defines a system component as a file that contains some information about the system. He notes that this information is encoded in some manner, but is not restricted to programming languages. Other possible encodings are higher level descriptions which may be used to generate code, and segments of documentation are also included in the definition of a system component. Ryman defines a package as a collection of parts. Both packages and parts are identified by a single name attribute.

4.3 System Representation

In the custom language approaches, modules and systems are characterized by the set of resources provided and the set of resources required. These sets define the interface of a module or system. The structure of the system and the manner in which the modules are incorporated into the system are closely related to the way that resources are passed between the modules.

Explicit Hierarchy (MIL 75, Cooprider, INTERCOL and NuMIL). These MILs model a system with an explicit hierarchy. MIL 75 uses a tree of subsystem nodes. Each node may have a single module associated with it to provide resources not provided by any of its children. All leaf nodes must have a module. INTERCOL and NuMIL represents a system as a directed acyclic graph whose leaf and internal nodes represent modules and configurations (subsystems) respectively. Cooprider's notation uses nested blocks, which form a tree with the modules as leaf nodes, and systems as the non-leaf nodes.

Directed Graph (Thomas). Thomas represents systems as directed graphs. Each graph has a designated node, called the distinguished node, whose interface characterizes the rest of the graph. Abstraction is provided by treating a subgraph graph as a single node with the same interface as its distinguished node. This abstraction mechanism provides a hierarchy similar to that used in the other custom language MILs. Each node may have several modules associated with it.

Set Theoretic Approach (Ryman). In this approach, the system is represented as a collection of sets, relations and restrictions on the values of the sets and relations. The entity relationship model [Chen 76] is used by Ryman to organize the system representation. The main entity sets are package and part. The uses relation is defined on the set part. This is the main distinction between Ryman and the MILs provided by the other authors. Modules do not use resources from other modules. Instead, resources use other resources.

4.4 Resource Access

The main purpose of a MIL is to arbitrate the access of resources between modules. In this section we examine how each of the surveyed MILs control the exchange of resources.

MIL 75. Resources in MIL 75 may be passed between siblings, or may be passed up (derived) or down (inherited) the system hierarchy. Parent nodes specify the resources that each child must provide, and any access that a child may have to the resources provided by sibling nodes. If a node has access to the resources provided by a sibling node, it has unrestricted use of all of the resources provided by that node.

Parents have access to all of the resources provided by their children and, by default, the children inherit access to the resources that have been granted to their parent. The parent can restrict the children's access to the parent's resources.

Thomas. Resources for a given node may be obtained from the modules associated with the node, inherited from other nodes that reference the given node or demanded from successor nodes. The resources obtained from these three sources are pooled into an environment in the node. These resources may be passed to to the nodes referencing the given node (*synthesized*), they may be used by the modules associated with the node, and they may be inherited by the successors. Access to cluster resources can be restricted to a subset of the operations that are defined on the type.

Cooprider. The interconnection level is used to arbitrate resource access in Cooprider's notation. He provides three methods of passing resources among subsystems. These are nesting, external systems and environments.

Several subsystems can be nested within a subsystem and resources may only be passed between the children and the parent. External subsystems can be explicitly named as the source of some subset of the subsystem's required resources. A subsystem can define an environment which provides resources to all nested subsystems. The resources in the environment are not only visible to the immediate children of the subsystem, but are visible to all of the descendants. Cooprider provides an invisible parent subsystem which encloses all of the top level subsystems; it provides a set of commonly used resources as an environment for all of the subsystems.

INTERCOL. INTERCOL provides two different hierarchies on the system. The first is the composition hierarchy comprised of modules and compositions. Resource access is governed by the interface of components and configurations.

The other is a block nesting of subsystems and modules that provides scope control over the visibility of modules and subsystems. A composition may only include other modules and compositions if they are visible.

NuMIL. Resource access in NuMIL is very similar to INTERCOL. Each module and system has an interface of provided and required resources that must be satisfied. The difference is that the required resources are specified by each implementation of a module, and by each configuration of a subsystem. Narayanaswamy and Scacchi claim that the resources required to implement other resources are attributes of the implementation, and not of the module family. As an example, consider a module that provides a queue resource. One implementation may use an array and not require any resources. Another implementation might use a linked list and require dynamic memory resources. Module and system families are only used to group modules and configurations that satisfy a common abstract interface.

Ryman. In Ryman's approach, resources use other resources. The relation package_uses(X,Y) is derived from the relation between resources (parts). As such, there are no rules on the access between resources.

4.5 Resource Typing

There is some disagreement among the MILs on whether the types of the resources should be specified in the MIL. The approaches can be divided into three groups: no types, abstract types and full typing.

No Types (MIL 75, Cooprider). DeRemer and Kron do not provide any means of specifying the types of resources. They state that resource typing is in the domain of programming-in-the-small. They do mention that the combined system will have to do as much bookkeeping as necessary to ensure that resources are used consistently between modules, but make no mention of how such a system should be implemented. Cooprider also leaves resource typing to the level of programming-in-the-small.

Abstract Types (Thomas,Ryman). Thomas only provides two types of resources, clusters and procedures. Cluster resources are used to implement an abstract data type. A cluster consists of a type, and operations that may be performed on that type. Procedure resources are the procedures from the level of programming-in-the-small. No method of sharing data resources is provided.

Ryman uses named types for resources. Support for object oriented programming is provided by providing subsets of resources that represent messages between modules and resources that represent data types. Several relations between the messages and the data type resources are used to indicate the types of parameters and results of messages.

Full Typing (INTERCOL,NuMIL). Tichy uses a resource-specification sublanguage to specify the type of resources. A clear distinction is made between the interconnection notation and the resource sublanguage. This allows the sublanguage to be installation dependent. The sublanguage used by Tichy is a modified subset of Ada.

He states that the interface must be complete. That is, if a resource is used in the definition of another resource, then it must be provided or required in the interface of that module. As an example, an abstract type may be used as a field in a record. The module that provides the record must also provide the abstract type or require it from another module.

In NuMIL, the interface is specified at two levels. The abstract level specifies the resource as a template defining its functional properties. Each implementation specifies the interface in the syntax of the implementation language. Since the resources are specified in the syntax of the implementation language, checking that the module versions satisfy the structural part of the concrete interface can be done by the development environment. The checks against the functional specification are somewhat more difficult. In the paper, several methods of showing that the module implements the functional specifications are suggested.

4.6 Resource Translation

If more than one language is to be used, then some method of ensuring consistency between languages is needed. We divide the approaches into four categories: not discussed, user defined, system defined and single language.

Not Discussed (MIL 75,Thomas). DeRemer and Kron mention that an advantage of using a separate MIL was the use of multiple languages for programming-in-the-small. However, they make no mention of how resource usage should be checked across language boundaries and state that the complete system should perform all static checks. Thomas also makes no mention of resource translation.

User Defined (Cooprider). Cooprider 's concrete level is used to obtain and manage the resources. Cooprider defines concrete objects as generalized files. Some of the things contained in these files are the system source, intermediate results, and the executable version of the system. Concrete objects are generated by rules that may use processors.

A processor is any program that produces a concrete object. It usually has one parameter (also a concrete object) and possibly a string giving processor options. Processors include compilers, code generators, linkers, and text formatters.

System Defined (INTERCOL). Tichy provides translators which translate the resource specification to that of the implementation language for a module. Tichy suggests that one less translator need be written if one of the implementation languages is used as the resource-specification sublanguage. The translators are invoked automatically by the development environment when systems are assembled.

Single Language (NuMIL,Ryman). NuMIL requires that the entire system be implemented in one language. Thus there is no need for resource translation. There is no mention by Ryman of the use of more than one language. The types of the resources in Ryman are specified at a high level of abstraction and a language model is used to map the high level representation to the implementation.

4.7 Environment and Enforcement

Various means of incorporating the MIL into the development environment are presented in the surveyed research. These methods can be divided into four groups: not discussed, separate tool, environment database, and logic systems.

Not Discussed (MIL 75). DeRemer and Kron do not discuss the manner in which MIL 75 is integrated into the development environment in any detail. They do mention that a more complicated compilation process and file system will be required to maintain all the information necessary to check the use of resources between modules.

Separate Tool (Thomas). Thomas gave much more consideration to the place that a MIL has in the development environment. He breaks the system construction task into four components. These are compilation, checking, binding, and linking. Each module is described by a *Description Unit* (DU) which describes the interface of the module. This interface is enforced by the compiler. The checking phase of the system construction process checks the consistency of the MIL description, and that the resources listed in the MIL for each node match the interface of the modules associated with the node. Binding uses the MIL description to resolve references to external resources.

Environment Database (Cooprider, INTERCOL, NuMIL). This approach incorporates the MIL into the development environment by introducing a database. Subsystem descriptions are entered into the database where they can be checked for consistency, in some cases before any implementation is done. Commands to construct the software can be issued to the database, which will invoke the necessary tasks.

Since the textual representation of resources are used in Cooprider's MIL, any enforcement of the system description must be done by the concrete-level processors. Logical candidates are the compilers, which are processors in Cooprider's system. This approach requires strongly typed languages such as Turing [Holt *et al.* 88] or Modula-2 [Wirth 85].

The INTERCOL environment handles the propagation of changes to any affected systems and manages the storage of derived versions and compositions. The consistency of all modules are checked at compile time, and the management of interface changes is automatically handled. Tichy requires the compilation process to examine the interface of a module and enforce the use of that interface.

The NuMIL environment provides strong support for enforcing the MIL specification. Source code for modules must be checked out before they can be modified. When the modules are checked in, they are tested by the environment to be compatible with their definition. Unlike some of the previous MILs, the compilers are not used to enforce the module interface. Thus the compilers need not be modified, and even languages with weak type constraints can be used.

Logic Systems (Ryman). The Prolog knowledge base is used by Ryman to build the design and development environment. Enforcement is provided by generating a single header file for each of the packages and by parsing the code back into the knowledge base. The application code that is parsed is the combination of the generated headers, and the hand written code. It is parsed using the language model to extract facts. These facts are compared with the design facts and any discreprencies are reported.

5. A Theory–Model Approach

This section presents a characterization of the MILs using the Theory Model paradigm. We proceed in two steps. The first is to provide a core theory that applies to all of the MILs. This theory is then extended for each of the MILs surveyed.

There are two conventions that we use throughout this section. The first is the lexical conventions for the sets and relations. The names of Entity sets have each word capitalised. Relation names consist of all lower case letters. In both cases, the names may consist of more than one word, and may contain spaces. The second convention is the use of a binary predicate name as an extension to an entity name. Given the binary predicate $p : X \times Y$, and the entity $a \in X$, the expression a.p denotes the set of all $b \in Y$ such that $(a,b) \in p$. That is, $a.p \equiv \{ b : Y \mid (a,b) \in p \}$. The same form is used to denote the value of an attribute of a given entity. Since the names of predicates and attributes are distinct, no conflict can occur.

5.1 Core Theory

From the proceeding sections that describe each of the MILs, we see that there are several core concepts. Resources are nameable entities from the domain of programming-in-the-small. Modules are written in some language for programming-in-the-small that may define some resources and may require some other resources. Subsystems are structural concepts used to group modules and to arbitrate access between modules.

Our core theory of MILs has six entity sets and five relations defined over the sets (see Figure 1). The six sets are *Component*, *Module*, *Group*, *Subsystem*, *Resource* and *Resource Reference*. The sets

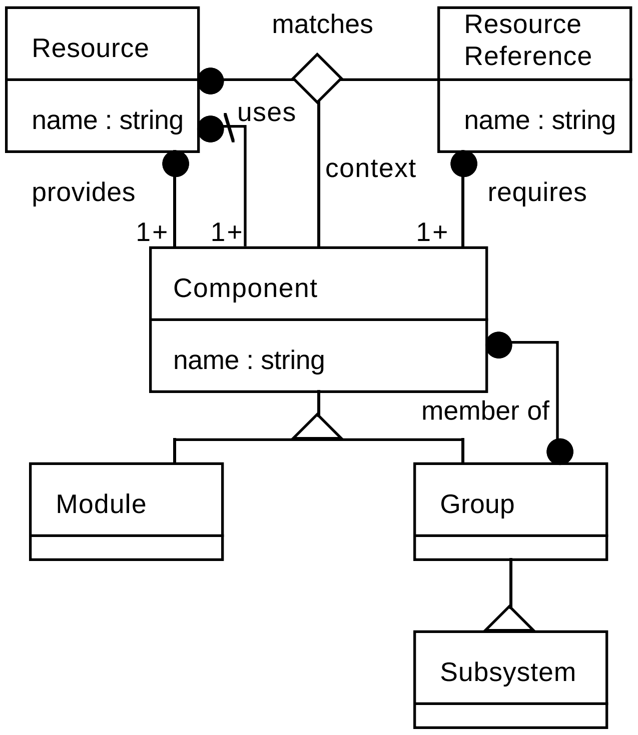

Figure 1 – ER Representation of Core Model

Module, Group and *Subsystem* are subsets of the set *Component*. The entity sets *Component*, *Resource* and *Resource Reference* have an attribute *name* of type string. The five relations are *provides, requires, matches, member of* and *uses*. These relations have the following signatures:

provides	:	Component ⟷ Resource
requires	:	Component ⟷ Resource Reference
matches	:	Component ⟶ (Resource Reference ⟷ Resource)
member of	:	Component ⟷ Group
uses	:	Component ⟷ Resource

The sets *Component*, *Module* and *Group* , and the relation *member of* model the structural interconnection provided by each of the MILs. Some MILs provide more than one structural framework over the modules. For example, Tichy's INTERCOL uses the subsystem hierarchy to provide scoping, and the composition hierarchy to provide system composition. Different structural interconnections will be modeled by subclasses of the entity set *Group*. Non-structural interconnection will be handled by extra relations.

The cardinality of the relation reflects that the interconnection may take different forms. One interconnection common to all of the MILs is the subsystem hierarchy and is modeled using the set *Subsystem*. This hierarchy provides scoping on the names of modules and resources. It also constrains the permissible interconnections between modules. We define two derived relations (not shown in the figure) as:

$$\text{scoped by} \equiv \text{member of} \cap (\text{Module} \times \text{Subsystem})$$
$$\text{child of} \equiv \text{member of} \cap (\text{Subsystem} \times \text{Subsystem})$$

The *scoped by(m,s)* relation is used to indicate that scope of the resources provided by the given module (*m*) is determined by the subsystem *s*. In most of the MILs, the definition of *m* is textually included in the definition of *s*. The *child of(s1,s2)* relation is used to handle the nesting of subsystems. The subsystem *s1* is a child of the subsystem *s2*. This is used to permit two subsystems to share a resource without the resource becoming generally available. We define the derived relation *parent of* as the inverse of the *child of* relation.

In all the MILs, components require resources by name. For some of the MILs this is not a problem since only one resource with a given name may exist in a given system. For others, more than one resource may have the same name, and scope resolution rules are used if they are both visible in the same context. We handle this by adding a separate entity type called *Resource Reference*, and a relation *matches* that associates it with the correct resource. The expression *matches(c,rr,r)* is a ternary relation that associates a resource reference with a resource in a given context. The axioms for each MIL will model the visibility rules as constraints on the *matches* relation. The cardinality constraint on the *matches* relation reflects the fact that a resource reference may not map to a resource in every component context, and in some cases may map to more than one resource in a given component context. The relation *matches* may be interpreted as a function that, given a component, returns a relation between resource reference and resource.

The interface of a module is modeled using the *provides* and *requires* relations. The relation *provides(c,r)* matches the component *c* with the resource *r* that it provides. Similarly, the relation *requires(c,rr)* associates a component *c* with the resource reference *rr* that it requires. The cardinality constraints on the *provides* and *requires* relations indicate that a component may provide zero or more resources and may require zero or more resource references. Not all components require resources, and some may require resources without providing any.

The axioms of our core model are expressed using the core ER schema as a framework. The first of these axioms is:

$$\forall\ r \in \text{Resource},\ rn \in \text{Resource Reference},\ c \in \text{Component}, \quad \text{matches}(c,rn,r) \Rightarrow rn.\text{name} = r.\text{name} \qquad \text{(C1)}$$

This axiom states that the value of the *name* attribute of resources and resource references associated by the *matches* relation must be the same. That is, components require resources by name. Since the component used as the context is the same that was used to enumerate the resource references, we defined the derived relation *uses* from *Component* to *Resource* that enumerates the actual resources required by the component.

$$\text{uses} \equiv \{\ \langle c,r\rangle \mid c \in \text{component},\ r \in \text{Resource},\ rr \in c.\text{requires},\ \text{matches}(c,rr,r)\ \} \qquad \text{(C2)}$$

The next three axioms deal with hierarchical structure of the MILs:

$$\exists\,!\ \text{root} \in \text{Subsystem} \ni \text{root.parent of} = \varnothing \qquad \text{(C3)}$$

$$\forall\ s \in \text{Subsystem} - \{\ \text{root}\ \},\ \text{child of}^{+}(s,\text{root}) \qquad \text{(C4)}$$

The first of these axioms (C3) state that there is a unique subsystem that is not a child of any other subsystem. We call this subsystem the root subsystem of the MIL description and refer to it as *root* in all other axioms. All of these axioms are for a MIL description of a particular system. We do not model MIL environments that may provide several subsystem libraries, which may or may not be used at a given time.

Axiom C4 states that all of the subsystems may be reached from the root by traversing the *child* relation. The *child of*$^{+}$ relation is the transitive closure of the *child of* relation. Axiom C5 requires that components may not provide and require resources with the same name. The last axiom, C6, states that only one module may provide a given resource. Other components may , in turn provide the resource, but it must originate at a single module.

$$\forall\ c \in \text{Component},\ c.\text{provide.name} \cap c.\text{requires.name} = \varnothing \qquad \text{(C5)}$$

$$\forall\ m1,m2 \in \text{Module},\ m1 \neq m2,\ m1.\text{provide} \cap m2.\text{provide} = \varnothing \qquad \text{(C6)}$$

The core theory just described is used by all of the MILs that we survey. There are four more axioms which are common to subsets of the MILs. Axioms EC1 and EC2 restrict the *child* relation to form a directed tree. Axiom EC3 limits each resource reference to at most one resource. The other axiom states that all leaf subsystems must have modules.

$\forall$ s $\in$ Subsystem, $\neg$ child of$^+$(s,s) (EC1)
$\forall$ s $\in$ Subsystem, | s.parent of | $\leq$ 1 (EC2)
|{ r $\in$ Resource | $\exists$ rr $\in$ Resource Reference, c $\in$ Component, matches (c,rr,r) } | $\leq$ 1 (EC3)
$\forall$ s $\in$ Subsystem, (s.child of = $\emptyset$) $\Rightarrow$ s.scoped by^{-1} $\neq$ $\emptyset$ (EC4)

We refer to the model formed by the ER schema and axioms C1 through C4 as the core theory. The model consisting of the core model and the extended axioms is called the standard theory. This relationship is shown in Figure 2.

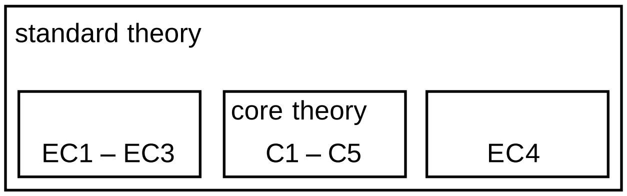

Figure 2 – Core Theories

Table 2 shows the relation between the core theory, standard theories, the two extended axioms and the MILs we have examined.

Standard Theory	Mil 75
	Cooprider
	NuMil
	Ryman
Core and EC1 - EC3	Intercol
Core and EC4	Thomas

Table 2 – Core Theories and MIL Models

From the core and standard models, it is possible to describe how each of the MILs extend this core model. Since we formalize all of the MILs in terms of these models, some of names of the relations must be changed to match our names. There are several classes of extensions we expect. The first is to the *provides* and *requires* relations. In the core model, these involve the union of the Module and Subsystem entity sets. Some of the MILs refine these relations, distinguishing between module and subsystem interfaces.

Additional relations may be provided. In the core model, there are only two relations, *provides* and *requires*, that expresses the interface of modules and systems. This corresponds to a single interface for a given module or subsystem that is accepted by the environment consisting of the parent and siblings of the module or subsystem. Some MILs have two interfaces for a component: one provided by the component, and the other expected by the environment. This requires more relations between the *Component* entity set and the *Resource* and *Resource Reference* entity sets.

There may be other relations, and other entity sets provided by the extended theories used to model concepts not present in the core model. The last extension that should be expected is the use of the extended relations and entity sets in restrictions on the *matches* relation. In the case of single resources with a given name, a one–to–one relation that ignores the context component of the relation is to be expected. The following sections present the extensions for each of the MILs in chronological order.

5.2 MIL 75

The MIL 75 theory is an extension to the standard theory. In MIL 75 all resource arbitration is between subsystems and DeRemer and Kron view modules as fragments of code that implement the resources originated by the subsystem. Some of the relations, such as the *originates* relation, provide information about the modules that implement the subsystems. In our model of MIL 75, these relations refer to the modules directly.

Resource arbitration in MIL 75 is half contractual. That is, both the parent and the child specify the resources that will be provided by the child. However, only the child specifies which resources it requires, which must be resolved with the resources provided by the subsystems to which it has been given access.

The *requires* relation has been split into two relations, the *requires derived* and *requires non-derived* relations. These relations correspond to the uses derived and uses nonderived statements in MIL 75. Derived resources are those provided by child subsystems. Thus the two sub-relations distinguish between resources provided by child subsystems, and resources provided by siblings or ancestor subsystems. The *requires* relation from the core theory is the union of the *requires derived* and *requires nonderived* relations.

Additional relations are the *originates*, *sibling access*, and *inherits restrict* relations. The *originates* relation describe the resources that are originated at the subsystem, and are defined in the module. The *sibling access* relation is defined by the parent of the subsystem and enumerate the sibling subsystems that resources may be obtained from. The *inherits restrict* relation is used by a parent subsystem to restrict the subsystems from which a child subsystem may inherit resources. In MIL 75, children subsystems inherit access to all of the systems that the parent has access to, unless the parent has specifically restricted the child. Just as the *uses* relation enumerates the actual resources that are required, two additional derived relations, *uses derived* and *uses nonderived*, enumerate the actual resources referred to by the *requires derived* and *requires nonderived* relations.

DeRemer and Kron do not discuss the issue of name space when presenting the MIL. While they have not explicitly prohibited more than one resource to have the same name, they also do not explicitly allow it. Since DeRemer and Kron insist on unique subsystem names, our model assumes that there is no support for having more than one resource with a given name.

Figure 3 shows the ER schema for MIL 75. The additional relations are shown as bold lines. The new constraints on core relations, such as the cardinality constraints on the *matches* and *scoped by* relations are expressed as axioms.

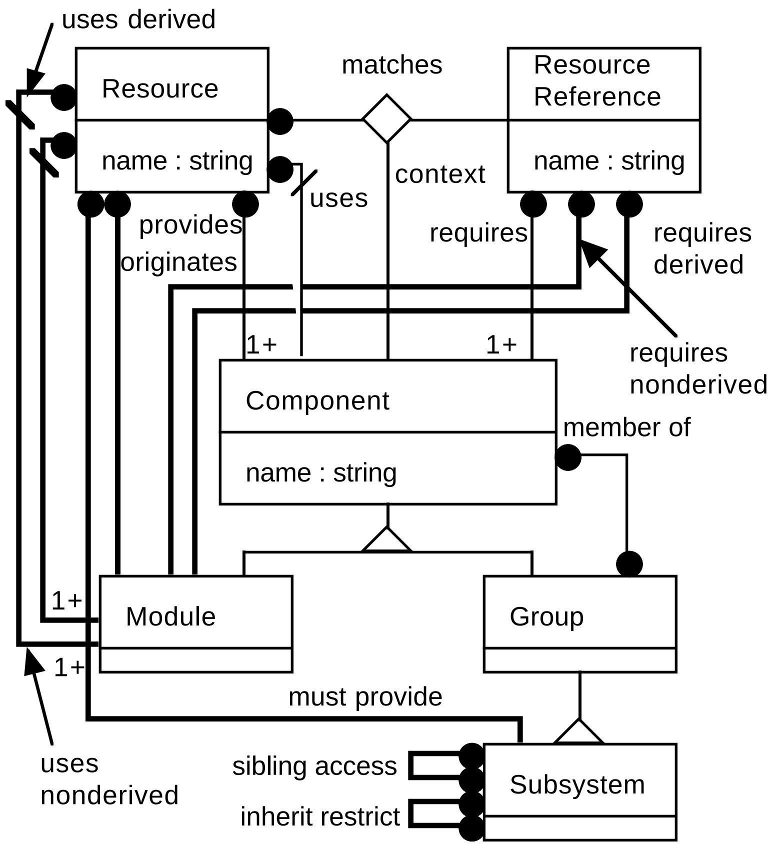

Figure 3 – ER Schema of MIL 75

The first two axioms of the extended theory, axioms D1 and D2, restrict the domains of the provides and requires relations to the *Subsystem* and *Module* entity sets. Axioms D3 and D4 restrict the cardinality of the *scoped by* and *matches* relations. There can be at most one module associated with a subsystem, and some subsystems only have subsystems as children. Axiom D5 flattens the resource reference space. That is, the resource denoted by a resource reference is the same in all contexts (see axiom C1). The last of this first group of axioms restricts the *requires* relation to be the union of the *uses nonderived* and *uses derived* relations.

$\text{dom(provides)} \subseteq \text{Subsystem}$ (D1)

$\text{dom(requires)} \subseteq \text{Module}$ (D2)

$\forall\, s \in \text{Subsystem},\ |\, s.\text{scoped by}^{-1} \,|\ \leq 1$ (D3)

$\forall\, r1, r2 \in \text{Resource Reference},\ r1 \neq r2 \Rightarrow r1.\text{name} \neq r2.\text{name}$ (D5)

$\text{requires} = \text{requires nonderived} \cup \text{requires derived}$ (D6)

Axioms D7 and D8 define the derivation of the uses derived and uses nonderived relations.

$\text{uses derived} \equiv \{ <c,r> \mid c \in \text{component},\ r \in \text{Resource},\ rr \in c.\text{requires derived},\ \text{matches}(c,rr,r) \}$ (D7)

$\text{uses nonderived} \equiv \{ <c,r> \mid c \in \text{component},\ r \in \text{Resource},$
$rr \in c.\text{requires nonderived},\ \text{matches}(c,rr,r) \}$ (D8)

DeRemer and Kron require that subsystem names be unique. In our extended theory this is expressed as:

$\forall\, s1, s2 \in \text{Subsystem},\ s1 \neq s2 \Rightarrow s1.\text{name} \neq s2.\text{name}$ (D9)

The next two axioms restrict the *must provide* relation. The first of the axioms states that the *must provides* relation is a subset of the *provides* relation. That is, the subsystem provides (the *provides* relation) at least those resources demanded by its parent (*must provide*). Axiom D11 states that the parent may not demand the same resource from more than one child subsystem.

$\forall\, s1 \in \text{Subsystem},\ s1.\text{must provide} \subseteq s1.\text{provides}$ (D10)

$\forall\, s1,s2,s3 \in \text{Subsystem},$
$(\text{child of}(s1,s3) \wedge \text{child of}(s2,s3) \wedge s1 \neq s2) \Rightarrow s1.\text{must provide} \cap s2.\text{must provide} = \emptyset$ (D11)

Axiom D12 requires that only one module in the entire system may originate a given resource. The derived relation *derives* is defined to be the union of all of the resources that the children must provide, and is given by axiom D13. The resources that a subsystem may provide to the rest of the system must be provided by the module associated with the subsystem, or must be provided by the children of the subsystem. This requirement is given by axiom D14.

$\forall\, m1, m2 \in \text{Module},\ m1 \neq m2,\ m1.\text{originates} \cap m2.\text{originates} = \emptyset$ (D12)

$\forall\, s1 \in \text{Subsystem},\ s1.\text{derives} \equiv \bigcup_{s2 \in s1.\text{child of}} s2.\text{must provide}$ (D13)

$\forall\, s1 \in \text{Subsystem},\ s1.\text{provides} \subseteq s1.\text{scoped by.originates} \cup s1.\text{derives}$ (D14)

Axioms D15 and D16 specify that resources from the *used derived* relation are derived resources and resources from the *uses nonderived* relation are provided by external subsystems.

$\forall\, s1 \in \text{Subsystem},\ s1.\text{uses derived} \subseteq s1.\text{derives}$ (D15)

$\forall\, r \in \text{Resource Reference},\ r \in s1.\text{uses nonderived} \Rightarrow$
$\exists\, n \in \text{Subsystem} \ni ((\text{sibling access}(s1,n) \vee \text{inherited access}(s1,n)) \wedge r \in n.\text{must provide})$ (D16)

The last axioms deal with restricting access to resources. Axiom D17 restricts the *sibling access* relation to sibling nodes. Axiom D18 limits the *inherit restrict* relation to ancestors, or siblings of ancestors; the implication is one way, since *inherit restrict* might be smaller.

$\forall\, s1, s2 \in \text{Subsystem},$
$(s1 \neq s2 \wedge \text{sibling access}(s1,s2)) \Rightarrow \exists\, s3 \in \text{Subsystem},\ \text{child of}(s1,s3) \wedge \text{child of}(s2,s3)$ (D17)

$\forall\, s1, s2 \in \text{Subsystem},$
$(s1 \neq s2 \wedge \text{inherit restrict}(s1,s2)) \Rightarrow$
$\text{child of}^{+}(s1,s2) \vee (\exists\, s3 \in \text{Subsystem},\ \text{child of}^{+}(s1,s) \wedge \text{child of}(s2,s3))$ (D18)

By default, a subsystem inherits all of the resources available to the parent. However, a parent may restrict the subsystems available to the child. This is given by the relation *inherit restrict*. For a given subsystem, this relation may be empty, implying full access, or an enumeration of the subsystems the child may access. We define a derived relation, *adr*, that lists those subsystems from which a subsystem may inherit resources. The first axiom, axiom D19, provides the definition of the *adr* relation. It is defined to be the empty for the root node (the first case), the systems given by the *inherit restrict* relation if it is not empty, or the systems inherited by the parent, and accessible siblings, if the *inherit restrict* relation is empty.

$$\forall\, s \in \text{Subsystem}, \qquad \text{(D19)}$$

$$s.\text{adr} \equiv \begin{cases} \emptyset & \text{if } \neg\,(\exists\, s' \ni \text{child of}(s,s')\,) \\ s.\text{inherit restrict} & \text{if } s.\text{inherit restrict} \neq \emptyset \\ s'.\text{adr} \cup s'.\text{sibling access} & \text{if } s.\text{inherit restrict} = \emptyset \end{cases}$$
$$\text{where } s' \in \text{Subsystem and child of}(s,s')$$

$$\forall\, s,s' \in \text{Subsystem},\ s.\text{inherit restrict} \neq \emptyset \wedge \text{child of}(s,s') \Rightarrow$$
$$s.\text{inherit restrict} \subseteq s'.\text{adr} \cup s'.\text{sibling access} \qquad \text{(D20)}$$

$$\forall\, s \in \text{Subsystem},\ r \in s.\text{uses nonderived} \Rightarrow$$
$$(\exists\, n \in \text{Subsystem} \ni (\text{adr}(s,n)\ \vee \text{sibling access}(s,n)\,) \wedge r \in n.\text{must provide}) \qquad \text{(D21)}$$

Axiom D20 limits the values of the *instr* relation to the subsystems to which the parent inherits access. This limits the subsystems a given subsystem may allow its children to inherit. The last restriction, axiom D21, restricts the *uses non–derived* relation to resources demanded of subsystems given by the *adr* relation. That is, the only resources that may be inherited are resources an ancestor demanded of the ancestor's child.

5.3 Thomas

The MIL proposed by Thomas is an extension to the core model and extended core axiom EC4. It uses a graph structure to describe the MIL; thus, the transitive closure of the *child* relation may be reflexive. A given subsystem may have more than one parent system.

In our model, subsystems are used to represent the nodes in Thomas's MIL. Thomas's subsystems are represented by a new class *Subgraph*. This new class is a subclass of *Group*, and the relations *member of* and *distinguished* is used to model the aggregate nature of Thomas's subsystems. The successors of nodes are given as part of the member of relation. We define a projection *successor* from this relation.

Our model does not associate the resources generated locally by a node with the modules that provide them. This can be derived from the *scoped by* relation which is used to associated modules with nodes (subsystems in our model) and the *provides* relation of each of the modules. The *generates locally* relation specifies the resources provided modules associated with the subsystem.

The two ternary relations *must synthesize* and *inherits from* are used to model the demands a node makes on successor nodes. The relation *must synthesize(g1,s2,rn)* indicates that the group *g1* must synthesize the resource reference *rn* to subsystem *s2*, and the relation *inherits from(g1,s2,r)* indicates that the group *g1* inherits resource *r* from subsystem *s2*. Figure 4 shows the ER diagram of our model of Thomas' MIL.

The *synthesize* and *must inherit* relations in Thomas's MIL are represented by the *provides* and *requires* relations of our core model. The derived relation *environment(s,r)* indicates that the resource *r* is an element of the environment of the subsystem *s*. Axiom To1 provides the successor projection. The successor relation is the *child* relation combined with the elements of the *member of* relation that define a subgraph as a member of a subsystem.

$$\text{successor} = \text{child} \cup (\text{member of} \cap (\text{Subgraph} \times \text{Subsystem})\,) \qquad \text{(To1)}$$

The next set of axioms are restrictions on the *matches* relation and the derivation rules for the relation *environment*. Since the MIL has no block structuring constructs, the resource given by a resource reference must be inherited from the parent node, provided by a successor node or provided by a module associated with the system. In the case of a module, the name of a resource is handled by the nodes

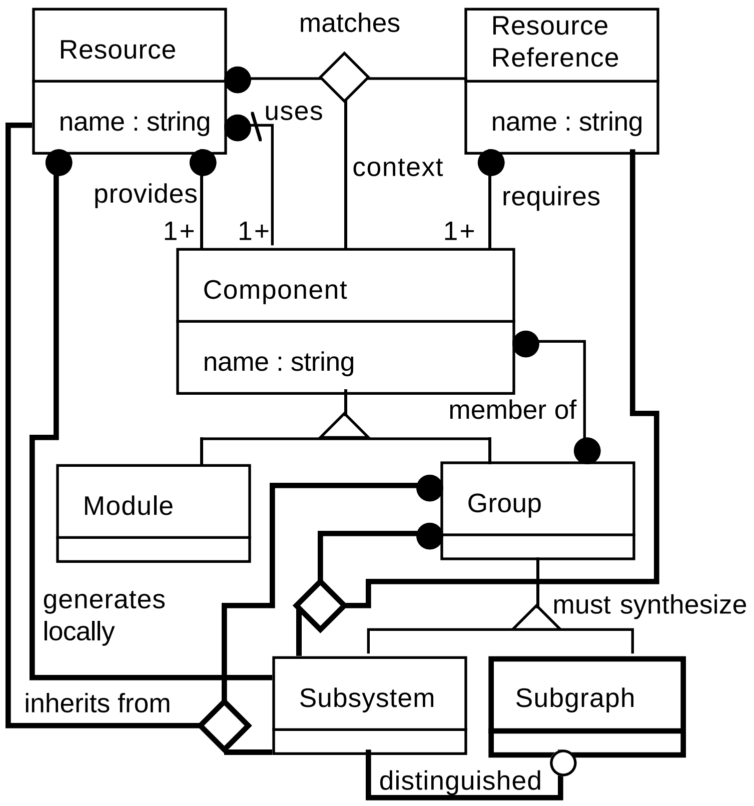

Figure 4 – ER Schema of Thomas's MIL

associated with the module. Thomas provides almost no discussion of resource reference conflicts. Since the scope of names is so restricted, it is doubtful if the issue arises. However, he does mention the split between resources provided by successor nodes and inherited from the parent. Our theory assumes that names are unique in their contexts. That is, the names of inherited resources will be distinct from the names of resources generated locally. Both will be distinct from names of resources provided by successor nodes. As axioms this assumption is:

∀ s ∈ Subsystem,
 inherited names(s) = { r : Resource Reference | requires(s,r) • r.name } (To2)

∀ s ∈ Subsystem,
 synthesized names(s) =
  { s2 : s.successor, rn : Resource Reference | must synthesize(s2,s,rn) • rn.name } (To2)

∀ s ∈ Subsystem,
 local names(s) = { m : s.scoped by | provides(m,r) • r.name } (To4)

∀ s ∈ Subsystem, inherited names(s), synthesized names(s) and
 local names(s) are pairwise disjoint (To5)

The derived relation *inherited names* is the set of names of resources that a subsystem requires from its parents. The relation *synthesized names* gives the net of names of resources that a subsystem demands from its successors. Finally the relation *local names* gives the set of names of resources provided by local modules. These three sets must be disjoint. Thus, the restrictions of the *matches* relation are:

$\forall$ c $\in$ component, rn $\in$ resource reference, (c,rn).matches = { r | r $\in$ Resources $\wedge$ r.name = rn $\wedge$
(c $\in$ Subsystem $\Rightarrow$
(($\exists$ s1 $\in$ c.successor $\ni$ r $\in$ s1.provides) $\vee$ ($\exists$ s2 $\in$ c.successor^{-1} $\ni$ inherits(c,s2,r)) $\vee$
($\exists$ m $\in$ c.scoped by $\ni$ r $\in$ m.provides)
) $\wedge$
c $\in$ Module $\Rightarrow$
(($\exists$ s1 $\in$ c.declares.successor $\ni$ r $\in$ s1.provides) $\vee$
($\exists$ s2 $\in$ c.declares.successor^{-1} $\ni$ inherits(c,s2,r)) $\vee$
($\exists$ m $\in$ c.declares.scoped by $\ni$ r $\in$ m.provides)
) $\wedge$
c $\in$ Subgraph $\Rightarrow$ matches(c.distinguished,rn,r)
) } (To6)

The environment of a node in Thomas' MIL is given by the axiom To7.

$\forall$ s $\in$ Subsystem,
s.environment $\equiv$ s.uses $\cup$s.generates locally $\cup$
{ r : Resource | $\forall$ s1 $\in$ s.successor, rn $\in$ Resource Reference,
must synthesize(s1,s,rn) $\wedge$ matches(s,rn,r) } (To7)

The next axiom states that any resources in the generates locally relation must be provided by modules associated with the subsystem. Axioms To9 and To10 require any subsystems named in the *must synthesize* or *inherits* relation to be successors of the subsystem. The last two axioms of this set requires that the distinguished node of a subgraph be a member of the subgraph, and that members of a given subgraph must be reachable from the distinguished node of the subgraph.

$\forall$ s $\in$ Subsystem, s.generates locally $\subseteq \bigcup_{p \in s.\text{scoped by}}$ p.provides (To8)
$\forall$ s $\in$ Subsystem, { s1 | $\exists$ r $\in$ Resource Reference $\wedge$ must synthesize(s1,s,r) } $\subseteq$ s.successor (To9)
$\forall$ s $\in$ Subsystem, { s1 | $\exists$ r $\in$ Resource $\wedge$ inherits(s1,s,r) } $\subseteq$ s.successor (To10)
$\forall$ t $\in$ Subgraph, t.distinguished $\subseteq$ t.member of (To11)
$\forall$ t $\in$ Subgraph, t.member of $\subseteq$ t.distinguished.successor* (To12)

Before a resource may be provided to the parent of a node or to successors of a node it must first be available in the environment of the node. Axioms To13 and To14 handle this.

$\forall$ s $\in$ Subsystem, s.provides $\subseteq$ s.environment (To13)
$\forall$ g1 $\in$ Group, s2 $\in$ Subsystem, (g1,s1) inherits $\subseteq$ s1.environment (To14)

Since this MIL uses contracts at both ends of the successor links (modeled by the successor relation), axioms To15 and To16 check to make sure that the contract is valid from both ends of the link. These axioms also ensure that all nodes that name the a node as a successor provide the required resources.

$\forall$ s $\in$ Subsystem,
{ c : s.successor | must synthesize(c,s,rn) $\bullet$ rn. name} $\subseteq$ { r : s.successor.provides $\bullet$ r.name} (To15)
$\forall$ s $\in$ Subsystem,
{ c : s.successor | inherits(c,s,r) $\bullet$ r.name } $\subseteq$ { r : s.successor.requires $\bullet$ r.name } (To16)

The last axiom, To17 ensures that the interface of a subgraph is the same as the interface of a distinguished node.

$\forall$ t $\in$ Subgraph,
t.requires=t.distinguished.requires $\wedge$ t.provides=t.distinguished.provides (To17)

5.4 Cooprider

Our model for the MIL proposed by Cooprider is an extension to the standard theory and is simpler than the previous two. We model concrete objects as modules and there are six new relations. The new relations are the *environment*, *req extern* , *env extern* , *req direct*, *env direct* and *env uses* relations. The *environment* relation models the definition of environment resources. The *req extern* and *env extern* model the specification of the subsystems that provide resources. Just as the *uses* relation enumerates the actual resources required, the *env uses* relation lists the environment resources.

Cooprider provides a mechanism to bypass the scope structure of the system description by specifying the module directly, even if it is not in the visible scope. For example, if there are two subsystems, A and B, and B has a subsystem C, a subsystem of A may require a resource directly from C by referring to it as B.C. This mechanism is modeled by the *req direct* and *env direct* relations. These relations are used in determining resource access, but there are no axioms restricting the membership in the relations. We do not model the construction rules given in the system description.

Figure 5 shows our model of the system representation provided by Cooprider's MIL. The cardinality constraints of the *environment* relation reflects the fact that a subsystem need not define an environment, or may define an environment with as many resources as it has available. The *req extern* and *env extern* relations the same cardinality constraints as the *requires* and *environment* relations.

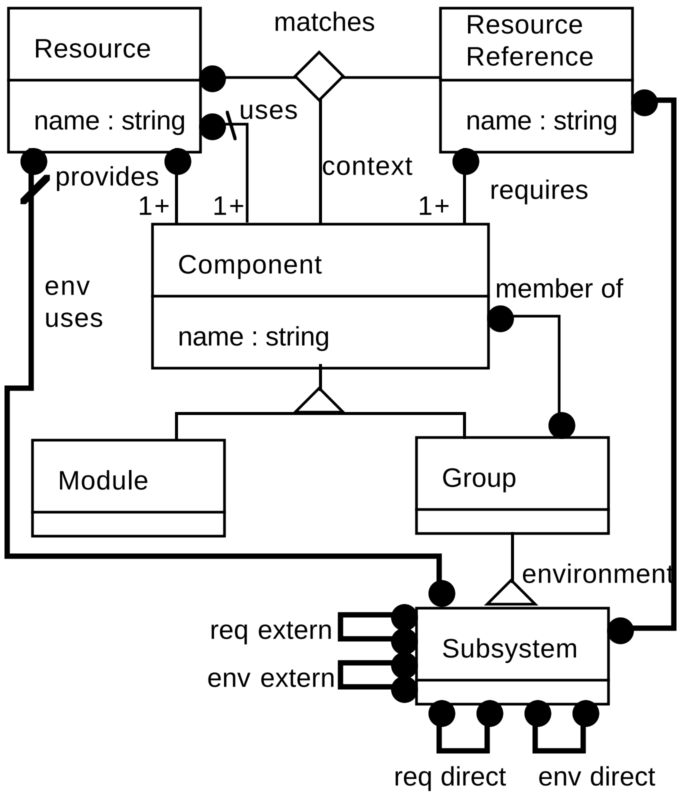

Figure 5 – ER Schema of Cooprider's MIL System Representation

The first axiom of the extended theory (Cp1) restricts the domain of the *requires* relation to the Subsystem entity class. Axiom Cp2 defines the *env uses* relation in a similar manner to the *uses* relation.

$$\text{domain(requires)} \subseteq \text{Subsystem} \qquad \text{(Cp1)}$$
$$\text{env uses} \equiv \{ <c,r> \mid c \in \text{component}, r \in \text{Resource}, rr \in \text{c.environment}, \text{matches}(c,rr,r) \} \qquad \text{(Cp2)}$$

Axiom Cp3 limits the resources a subsystem may require to the resources required by the parent, the resources in the environment defined by its ancestors and provided by named external subsystems and the resources provided by children. The named external subsystems must be a child of an ancestor to be visible. Cooprider's MIL does not state the resources required for a given module. The required resource of the subsystem it implements provides a pool of resources on which it can draw.

$$\forall\, s \in \text{Subsystem},\ s.\text{uses} \subseteq s.\text{parent.uses} \cup \bigcup_{p \in s.\text{parent of}^{+}} p.\text{environment} \cup \bigcup_{e \in s.\text{req extern}} e.\text{provides} \cup \bigcup_{c \in s.\text{child of}} c.\text{provides} \quad (\text{Cp3})$$

Axiom Cp4 states that the resources added to the environment by the *environment* relation must be provided by the subsystems enumerated in the *env extern* relation. Axiom Cp5 states that resources provided by a subsystem are provided by its children or by the concrete objects (modules) implementing the subsystem.

$$\forall\, s \in \text{Subsystem},\ s.\text{env uses} \subseteq \bigcup_{e \in s.\text{env extern}} e.\text{provides} \quad (\text{Cp4})$$

$$\forall\, s \in \text{Subsystem},\ s.\text{provides} \subseteq \bigcup_{c \in s.\text{child of}} c.\text{provides} \cup \bigcup_{m \in s.\text{scoped by}} m.\text{provides} \quad (\text{Cp5})$$

Axiom Cp6 provides the restrictions on the *matches* relation. If a resource reference is visible in a given context, the resource it maps to is the resource with the same name provided by a subsystem that is named in an environment clause or a requires clause of an ancestor subsystem. The previous axiom (Cp5) made sure that the resource is ultimately provided by a module. The requirement that only one resource with the same name can be visible in a given context is provided by the cardinality constraint on the *matches* relation.

$$\begin{aligned}&\forall n \in \text{Resource Reference},\ s \in \text{Subsystem},\\ &\quad (n,s).\text{matches} = \{\, r\text{: Resource} \mid r.\text{name} = n.\text{name} \wedge\\ &\qquad \exists\, s1 \in \text{Subsystem} \ni (\text{provide}(s1,r) \wedge (s1 \in (\bigcup_{p \in s.\text{parent of}^{+}} p.\text{env extern}) \vee s1 \in (\bigcup_{p \in s.\text{parent of}^{+}} p.\text{req extern})))\} \quad (\text{Cp6})\end{aligned}$$

The last three axioms handle environment resources and the visibility constraints on external subsystems in requires and environment clauses. Axiom Cp7 states that any resources added to the environment by a module must first be required by the module. Axiom Cp8 defines a derived relation *block visible* that defines the block scoping rules. Axioms Cp9 and Cp10 state that any subsystems named in the requires and environment clause must be visible by block scope rules. They must be a sibling of an ancestor of the subsystem.

$$\forall\, s \in \text{Subsystem},\ s.\text{environment} \subseteq s.\text{requires} \quad (\text{Cp7})$$
$$\text{block visible}(s1,s2) \equiv \{\, \langle s1,s2 \rangle \mid s1,s2 \in \text{System},\ \exists\, s3 \in s1.\text{parent of}^{+} \ni \text{child of}(s2,s3)\,\} \quad (\text{Cp8})$$
$$\forall\, s1,s2 \in \text{Subsystem},\ \text{req extern}(s1,s2) \Rightarrow \text{block visisble}(s1,s3) \quad (\text{Cp9})$$
$$\forall\, s1,s2 \in \text{Subsystem},\ \text{env extern}(s1,s2) \Rightarrow \text{block visisble}(s1,s3) \quad (\text{Cp10})$$

5.5 INTERCOL (Tichy)

As with Cooprider’s notation, we do not directly model the version and revision control provided by their notations. However, some structural entities are needed to handle the version and revision control. These entities are included in our model, but the restrictions on the entities needed to model versions and revisions are not given. The INTERCOL model is an extension to the core theory and Extended Axioms EC1–EC3.

INTERCOL does not require that modules be the leaves of the architecture model. Members of a system family are represented by compositions. The compositions group systems and modules that are children of the given system, or children of ancestor systems. Thus, it is possible to have a subsystem that only has compositions, and those compositions group the siblings of the given system. All subsystems must have at least one composition. The required resources of the elements of the compositions are resolved between elements of the composition, and the required resources of the system of which the composition is a member. The implication of this approach is that the *matches* relation of our model is interpreted relative to the system that contains the composition, not the system that requires the resource. This will be shown in the axioms.

We do not model the full resource specification sublanguages. All of the resource sets have been unbundled and the *provides* and *requires* relations enumerate all resources and subresources provided and required.

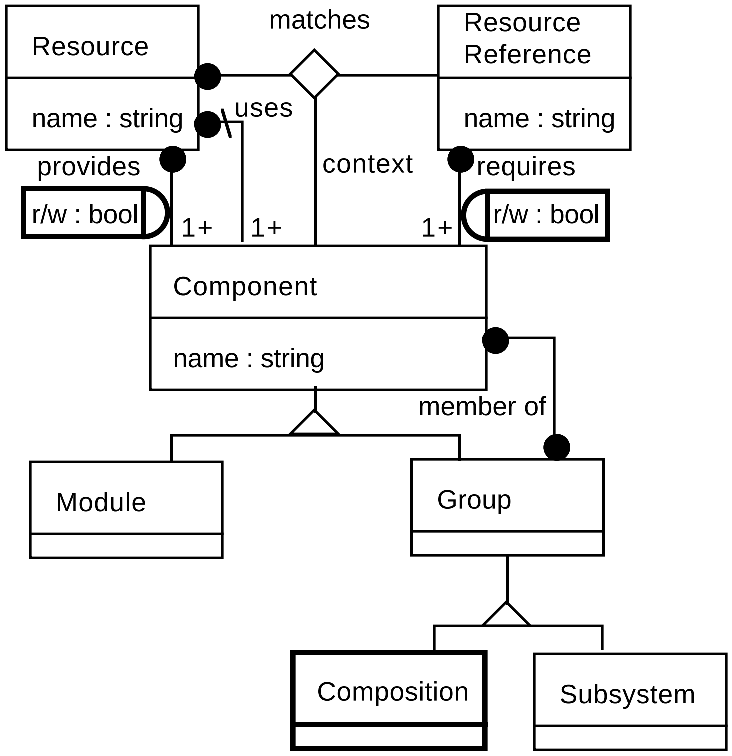

Figure 6 – ER Schema of INTERCOL System Model

Figure 6 shows our ER representation of the system model used by INTERCOL. Since INTERCOL module families are grouped together in a single syntactic unit called module, and since all implementations have the same interface as the module family, we use a single entity set, *Module*, to represent the module family.

System families in INTERCOL consist of module families, system families and compositions. The module and system families are building blocks for the compositions. System families are represented using the entity set *Subsystem*. The nested module and system family building blocks are indicated using the *scoped by* and *child* relations. Some restrictions may apply to both types of components. For these restrictions, we define the derived relation *building block* as follows:

building block $\equiv$ child of $\cup$ scoped by (T1)

Two other projections of the *member of* relation are of interest: *is comp* and *given el*. The derived relation *is comp(c,s)* is used to indicate that *c* is a composition of the subsystem *s*. The cardinality constraint on the relation is expressed as an axiom and indicates that every subsystem must have one or more compositions. In the case of default compositions, one is created and related to the appropriate subsystem. Axiom T2 defines the is comp derivation, and axiom T3 gives the cardinality constraint on the derived relation.

is comp $\equiv$ (Composition $\times$ Subsystem) $\cap$ member of (T2)
$\forall$ s $\in$ Subsystem, s.is comp $\neq \emptyset$ (T3)

The derived relation *given els* is used to enumerate the elements of a composition. The element *given els(cp,co)* indicates that component *cp* is an element of the composition *co*. It is empty for the default compositions. The derived relation *element of* is used in all of the constraints. It has the same signature as the *given els* relation and consists of the *given el*s relation for specified compositions and all of the building block components for default compositions. Specifically:

given els $\equiv$ (Component $\times$ Composition) $\cap$ member of (T4)

element of ≡ given els ∪ { (cp,co) | co ∈ Composition, cp ∈ Component ∋
co.given el = ∅ ∧ cp ∈ co.is_comp^{-1}.building block $^{-1}$ } (T5)

We abstract out version control by using *Component* as the co-domain of the *given els* and *element of* relations. Since all members of the family are constrained to have the same interface, we do not need to model which element is used. If needed, this could be added as an association attribute, and axioms added to deal with the attribute.

We have added a boolean attribute to the *provides* and *requires* relations, which is true if a particular resource has been provided or required writeable and false if it is read only.

Four entity sets and one relation have been introduced to approximate the typing system used in INTERCOL where every resource has a type. We do not attempt to model all of the details of the type system such as the order and types of parameters in a procedure. We simplify the type system to use two relations. The first, *involves*, indicates that one resource involves another resource as part of the type of the resource. The other relation is used to determine if two types are compatible. An example is that the type a procedure resource provided by a module may have more parameters than the type of the same resource provided by a composition. This is allowed provided that the extra parameters have default values. Figure 7 shows the new entity sets and the relation used to model the type system.

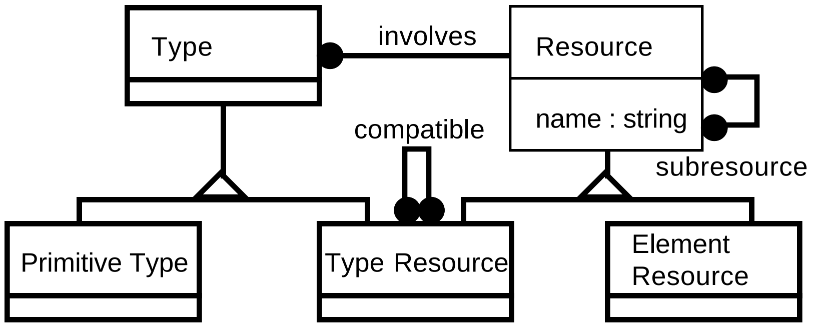

Figure 7 – ER Schema of Types in INTERCOL

The new entity sets are *Type*, *Primitive Type*, *Type Resource* and *Element Resource*. The sets *Type Resource* and *Element Resource* are subsets of the *Resource* set. Type resources are resources that represent types of entities, such as record resources or the resource that represents the signature of a procedure or function. Element resources are resources that represent the data and control entities of the software system, such as the procedures and variables.

The entity set *Type* is used to model types of resources. It has two subtypes, *Primitive Type* and *Type Resource.* The set *Primitive Type* is used to model primitive types such as integer and character, and is used to ground the type system. Modules and subsystems need not import primitive types.

The relation *involves(r,t)* indicates that used of the resource *r* involves the use of the type *t*. For element resources, the associated types are the type of the resource. In the case of type resources, the associated types are the types of subelements of the resource. An example is the resources used for types of parameters of a procedure signature, or the types of the fields in a record type.

The *compatible* relation is used to handle the case when the type provided is richer than the type required. In this case, the resource has two associated type resources which must be compatible. This relation is not necessarily symmetric. The relation *compatible(r1,r2)* indicates that the provided resource *r1* is compatible with the required resource *r2*. The only constraint (axiom T6) we put on the *compatible* relation is that it must be reflexive. Another constraint that is subsumed into the other axioms is that any subresource of an element resource must also have compatible type resources.

∀ tr ∈ TypeResource, compatible(tr,tr) (T6)

INTERCOL provides a block structured approach. The name of a provided resource is visible if the component is at the same level, or is the child of an ancestor. This *matches* relation is not used the same as in the other MILs. Instead, it will be used only when comparing components of a composition.

Resources provided by building blocks are also visible. We restrict the *matches* relation with the following axiom:

$\forall$ c $\in$ Component, rn $\in$ Resource Reference (c,rn).matches = findres(c,rn) (T7)

where:

$$\text{findres(c,rr)} = \begin{cases} \{r\} & \text{if r.name = rr.name} \wedge \exists \text{ c1} \in \text{c.building block} \ni \text{provide (c1,r)} \\ \text{findres(c2,rr)} & \text{if } \exists \text{ c2} \in \text{c.building block}^{-1} \\ \{\} & \text{otherwise} \end{cases}$$

Resources required by a subsystem may not be provided by any of the elements of compositions of that subsystem. This is modeled by axiom T8.

$\forall$ s $\in$ Subsystem, cs $\in$ Composition, cp $\in$ Component,
(is comp(cs,s) $\wedge$ element(cp,cs)) $\Rightarrow$ s.requires.name $\cap$ cp.provides.name = $\varnothing$ (T8)

Only one element of a composition provides a given resource, and any resources provided by a subsystem must be provided by an element of the composition. The is modeled by axioms T9 and T10.

$\forall$ c1, c2 $\in$ Component, co $\in$ Composition,
(element(c1,co) $\wedge$ element(c2,co)) $\wedge$ c1 $\neq$ c2 $\Rightarrow$ c1.provides $\cap$ c2.provides = $\varnothing$ (T9)

$\forall$ s $\in$ Subsystem, c $\in$ Composition, is comp(c,s) $\Rightarrow$ s.provides $\subseteq \bigcup_{\text{cp} \in \text{c.elements}}$ cp.provides (T10)

Resources required by the system is a superset of the resources required by elements less the resources provided by the elements. This restriction is handled by axiom T11. Notice that the two given elements of the has expression are the components and the resource. The result is the set of resource references associated with the provided resources by the components.

$\forall$ s $\in$ Subsystem, c $\in$ Composition,

is comp(c,s) $\Rightarrow$ s.requires $\supseteq \bigcup_{\text{cp} \in \text{c.elements}}$ cp.requires – $\bigcup_{\text{cp} \in \text{c.elements}}$ (cp,cp.provides).matches (T11)

As with the derivation of the *matches* relation, INTERCOLis a block structured language. A component may only be a member of the composition if it is a building block in the same subsystem, or a visible building block in an enclosing subsystem.

$\forall$ s $\in$ Subsystem, co $\in$ Composition, cp $\in$ Component, (T12)
is comp(co,s) $\wedge$ element(cp,co) $\Rightarrow$ building block(cp,s) $\vee$
($\exists$ s2 $\in$ Subsystem $\ni$ child^{+}(s,s3) $\wedge$ building block(cp,s))

All resources in INTERCOL are typed. If the type of the resource is another resource (not a primitive type), then the other resource must also be provided or required by the component. In both cases, the relation *matches* is evaluated relative to the system containing the composition.

$\forall$ c $\in$ Component, r1, r2 $\in$ Resource, s $\in$ Subsystem, co $\in$ Composition
(r1 $\in$ c.provides $\wedge$ r1 involves r2 $\wedge$ r2 $\in$ Type Resource) $\Rightarrow$
$\exists$ r3 $\in$ Resource, compatible(r2,3) $\wedge$ (r3 $\in$ c.provides $\vee$
((is comp(co,s) $\wedge$ element(c,co)) $\Rightarrow$ r3 $\in$ (s,c.requires).matches)) (T13)
$\forall$ c $\in$ Component, r1, r2 $\in$ Resource Reference, s $\in$ Subsystem, co $\in$ Composition,
((is comp(co,s) $\wedge$ element(c,co)) $\wedge$
(r1 $\in$ c.requires $\wedge$ (s,r1).matches.involves (s,r2).has $\wedge$ (s,r2).matches $\in$ Type Resource)) $\Rightarrow$
(($\exists$ r3 $\in$ Resource Reference, compatible((s,r2).matches,(s,r3).has) $\wedge$ r3 $\in$ c.requires)
$\vee$ ($\exists$ r4 $\in$ Resource, compatible((s,r2).matches,r4) $\wedge$ r4 $\in$ c.provides)) (T14)

INTERCOL allows designers to mark resources as read only. In our model, this is represented by the read write attributes on the *provides* and *requires* relation. This attribute is true if the resource is writeable. Subresources of read only resource must also be read only. The *subresource(r1,r2)* relation indicates that *r1* is a subresource of *r2*.

$\forall$ c1,c2 $\in$ Component, r1$\in$ Resource, r2 $\in$ Resource Reference, s $\in$ Subsystem, co $\in$ Composition,
(r1 $\in$ c1.provides $\wedge$ $\neg$(r1,c1).provides.r / w) $\Rightarrow$
((is comp(co,s) $\wedge$ element(c2,co) $\wedge$ r2 $\in$ c2.requires $\wedge$ (s,r2).matches = r1)$\Rightarrow$
$\neg$ (r2,c2).requires.r / w)) (T15)

$\forall$ c $\in$ Component, r1,r2 $\in$ Resource,
(provides(c,r1) $\wedge$ provides(c,r2) $\wedge$ subresource(r2,r1) $\wedge$ $\neg$(r1.c).provides.r / w) =>
($\neg$ (r2,c).provides.r / w) (T16)

5.6 NuMIL (Narayanaswamy and Scacchi)

We model NuMIL as an extension to the standard core theory. The main components of NuMIL are modules and configurations. Module families and subsystem families are used only to provide a logical grouping to the modules and configurations. Configurations may name modules and other configurations as members. A module family in a configuration is a short form for all of the configurations generated by the members of the module family. In our theory we assume that this has been expanded to the members of the module family.

We use the entity set Subsystem to represent configurations. The relations *scoped by* and *child* are used to model membership of elements in a configuration. Like Tichy, the bindings of resource references is dependent on the use of the element within a configuration. Unlike Tichy, NuMIL does not provide block scoping for names of modules, subsystems or resources. This simplifies the restrictions on the *matches* relation. Our ER schema for NuMIL is shown in Figure 8.

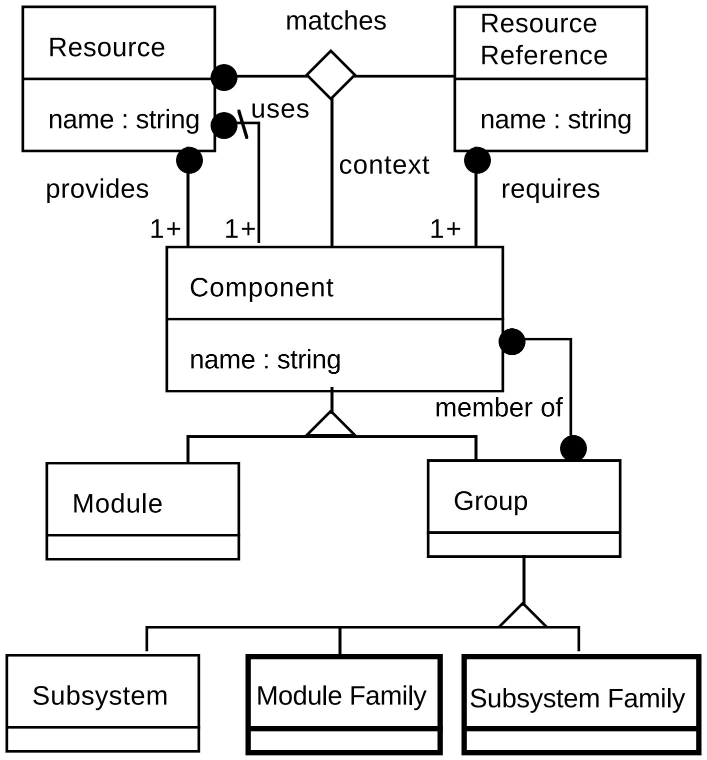

Figure 8 – ER Schema of NuMIL System Model

Our model of NuMIL has two extra entity sets and one extra relation. The two entity sets are *Module Family* and *Subsystem Family*. Both sets are subclasses of the set *Component*. NuMIL configurations are represented by the entity set *Subsystem*. The new relation, *family member of*, is used to model the relation between modules and module families and between configurations and subsystem families. The first axiom we examine provides the definition of the *family member of* relation. As with Tichy, we define the building block relation to deal with restrictions that apply to both the *child* and *scoped by* derived relations.

$$\text{family member of} \equiv \text{member of} \cap ((\text{Module} \cup \text{Subsystem}) \leftrightarrow (\text{Module Family} \cup \text{Subsystem Family})) \qquad (N1)$$

$$\text{building block} \equiv \text{child} \cup \text{scoped by} \qquad (N2)$$

Axioms N3 and N4 restrict the *family member of* relation to relations between module and module family and between subsystem and subsystem family. The first two axioms state that only modules may be member of module families and that only subsytems may be member of subsystem families.

$$\text{domain}(\text{family member of} \rhd \text{Module Family}) \subseteq \text{Module} \qquad (N3)$$
$$\text{domain}(\text{family member of} \rhd \text{Subsystem Family}) \subseteq \text{Subsystem} \qquad (N4)$$

The next axiom, N5, states that members of a module family must provide at least the resources provided by the module family. Axiom N6 states that the members of a subsystem family must provide at least the resources provided by the subsystem family.

$$\forall\ m \in \text{Module},\ mf \in \text{Module Family},\ \text{family member of}(m,mf) \Rightarrow mf.\text{provides} \subseteq m.\text{provides} \qquad (N5)$$
$$\forall\ m \in \text{Subsystem},\ sf \in \text{Subsystem Family},\ \text{family member of}(s,sf) \Rightarrow sf.\text{provides} \subseteq s.\text{provides} \qquad (N6)$$

The rules of family membership for modules and configurations is not consistently presented by Narayanaswamy and Scacchi. One paper [Narayanaswamy & Scacchi 87a]states that the required resources are not part of the module family template. It does not define the interface of subsystem families and refers to the other paper for the formal definition of software families. The other paper [Narayanaswamy & Scacchi 87b] states that both module and system families may require resources. The issue is not critical since the building blocks of configurations are other configurations and modules, and the families are used only to group related configurations and modules.

The next three axioms deal with well formed configurations. Axiom N7 limits the resources provided by a configuration to the resources provided by its members. Axiom N8 indicates that the same resource may not be provided by more than one element of a configuration. The last axiom, axiom N9, limits the required resources to the resources required by the members of the configurations but not provided by any other element of the configuration. This definition is different from the one provided by Tichy, who requires the differences between the two unions to be a subset of the resources required by the configuration family, since all configurations share the same interface. INTERCOL's approach also allows the person specifying the system some leeway in providing more resources than the configuration actually needs without invalidating the system description.

$$\forall\ s \in \text{Subsystem}\ s.\text{provides} \subseteq \bigcup_{c \in s.\text{building block}} c.\text{provides} \qquad (N7)$$

$$\forall\ s \in \text{Subsystem},\ c1,c2 \in s.\text{building block},\ c1 \neq c2 \Rightarrow c1.\text{provides} \cap c2.\text{provides} = \varnothing \qquad (N8)$$

$$\forall\ s \in \text{Subsystem},\ s.\text{uses} = \bigcup_{c \in s.\text{building block}} c.\text{uses} \ - \bigcup_{c \in s.\text{building block}} c.\text{provides} \qquad (N9)$$

Like Tichy, the *matches* relation in NuMIL must be specified relative to the use of the module in a configuration. Thus axiom N10 is a simple modification of axiom T7.

$$\forall\ c \in \text{Component},\ rn \in \text{Resource Reference}\ (c,rn).\text{matches} = \text{findres}(c,rn) \qquad (N10)$$

where:

$$\text{findres}(c,rr) = \begin{cases} \{r\} & r.\text{name} = rr.\text{name} \wedge \exists\ c1 \in c.\text{building block} \ni \text{provide}\ (c1,r) \\ \text{findres}(c2,rr) & \exists\ c2 \in c.\text{parent} \\ \{\} & \text{otherwise} \end{cases}$$

Narayanaswamy and Scacchi state that the matching the types of the resources builds upon the method used by Tichy. The additions to this method involve the use of pre and post conditions on each of the functional resources, and restrictions on these. Since we are interested in the structural aspects, the model we and the axioms used for INTERCOL can be used for this model.

5.7 Ryman

The main difference between the MIL system proposed by Ryman and the previous MILs is that Ryman's system does not require a particular set of constructs in the MIL. Instead, the developer first specifies the design theory, and then produces a design model (facts) that specifies the target system. In this paper we review the particular design theory (MIL) Ryman gives.

Since we are using Ryman's approach to the theory model paradigm to model the system representation used by the MILS, the translation in this case is simple. We rename some of the entity sets and relations of his theory to match the names we use in our core model. There are several differences between the theory Ryman uses and our core model. The first is that Ryman's model uses a flat name space, with global names for modules and resources. The second is that resource usage is arbitrated at the resource level. That is, resources use other resources. Modules are used only to package the resources, and the uses relation between modules is a derived relation based on the usage of its parts.

Figure 9 shows the ER schema for the system representation used by Ryman. It is a translation of the ER schema used by Ryman [Ryman 89], but using our notation. The entity sets *Resource* and *Module* represent the entity sets *parts* and *packages* in Ryman's notation. By making Resource a subclass of component, the *requires* relation, in conjunction with the *matches* relation may be used to represent Ryman's *uses* relation. The *matches* relation is constrained to be a one to one relation in all contexts. Our derived *uses* relation now corresponds to Ryman's *uses* relation. The derived relation *pkg uses* is the same as that used by Ryman. Our *provides* relation is equivalent to Ryman's *contains* relation.

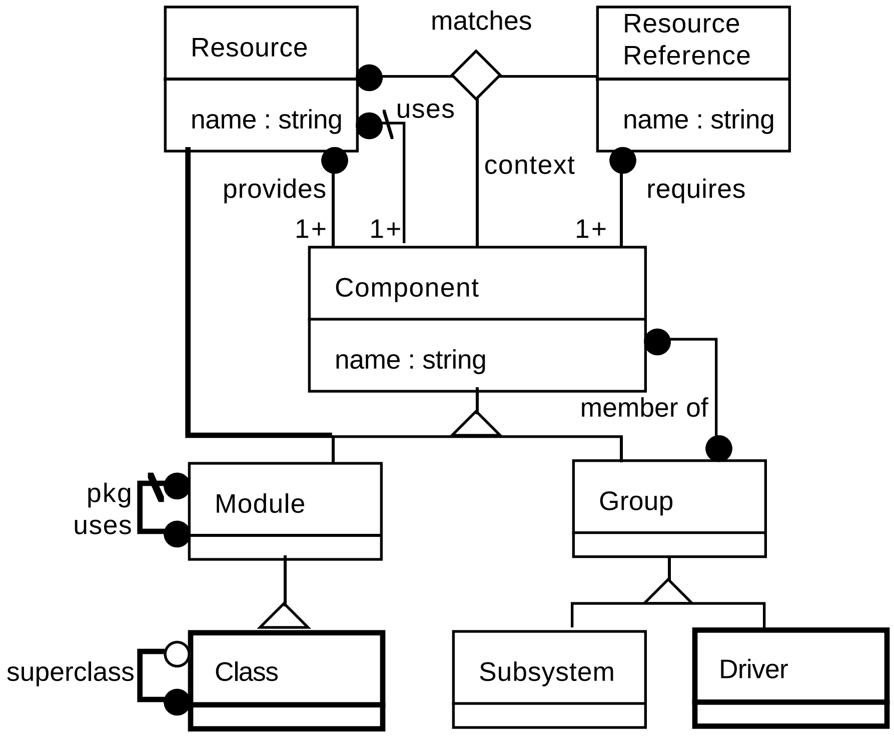

Figure 9 – ER Schema of Ryman's System Representation

The entity set Driver is a subclass of the Group entity set. The *member of* relation subsumes the *includes* and ≤ relations in Ryman's model. We define the following projections:

included in ≡ member of ∩ ((Resource ∪ Module) × Driver) (R1)

≤ ≡ member of ∩ (Driver × Driver) (R2)

The entity set *Class* and the relation *superclass* are used to model object oriented aspects of the design.

Figure 10 shows the ER schema representing the types of elements in the system. The model is taken directly from Ryman. The two relations *ptype* and *rtype* were given no name by Ryman since they were combined into ternary predicates for use in Prolog.

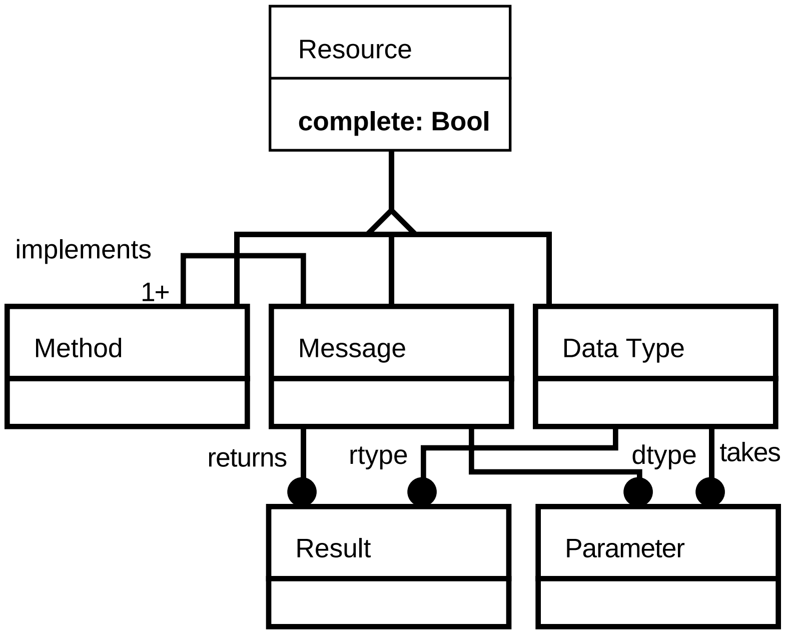

Figure 10 – ER Schema of types in Ryman's System

Several axioms are needed to make sure the model conforms to core theory. These first identifies a single unique system, SYS, and indicates all modules are scoped by this single system. The *child* relation is empty since there is only one subsystem. The last of these axioms makes sure that all modules are members of the single subsystem.

Subsystem = { "SYS" } (R3)
child = ∅ (R4)
dom(member of) = Module (R5)

The other axioms are a direct translation from Ryman. The first two of these axioms guarantee the existence of certain entities. The first indicates that there is a unique module called "OOP". The second, R7, indicates that there is a unique class with the name "Object". These are used to model object oriented programming. The next three axioms indicate that "Object" has no superclass, that every other class has a superclass, and that the transitive closure of the *superclass* relation is not reflexive. That is, there are no cycles in the superclass relation.

∃! m ∈ Module ∍ m.name = "OOP" (R6)
∃! c ∈ Class ∍ c.name = "Object" (R7)
"Object".superclass = ∅ (R8)
domain(superclass) = Class – { "Object" } (R9)
∀ c ∈ Class ∍ ¬ superclass$^+$(c,c) (R10)

A restriction Ryman places on the names of methods is that the name is a concatenation of the class and message names. This is handled by the axiom R11. The function *concat* returns the concatenation of two strings.

∀ m ∈ Method, ms ∈ Message, c ∈ Class,
(provides(c,m) ^ implements(m,ms)) ⇒ method.name = concat(c.name,ms.name) (R11)

The last five axioms deal with packaging the parts (resources) into drivers. The first of these axioms states that the ≥ relation is a partial order on the entity set *Driver*. The relation represents the order of detailed design of the driver. The next two relations indicate that all modules must be included by a driver, and that if a resource is included in a driver other than the one that includes the associated module, that the driver including the resource is defined previous to the other driver. The last two axioms constrain the complete attribute of a resource to be true if the resource has been included in a driver.

$\geq$ is a partial order on Driver (R12)

$\forall$ m $\in$ Module, $\exists$ d $\in$ Driver $\ni$ included in(m,d) (R13)

$\forall$ r $\in$ Resources, m $\in$ Modules, d1,d2 $\in$ Drivers,
(provides(m,r) $\wedge$ included in(r,d1) $\wedge$ included in(m,d2)) $\Rightarrow$ d2 $\geq$ d1 (R14)

$\forall$ r $\in$ Resources, d $\in$ Driver includes(d,r) $\Rightarrow$ r.complete = true (R15)

$\forall$ m $\in$ Modules, d $\in$ Driver, r $\in$ Resource, (included in(m,d) $\wedge$ provides(m,r)) $\Leftrightarrow$
r.complete = true (R16)

6. Comment and Conclusions

This exercise revealed two soft spots in the discussions of MILs. The first has to do with the binding of names to resources. Many of the MILs used phrases such as “a module cannot provide and require the same resource.” However, since the resource binding is not known until the module is used in most systems, this cannot be checked independent of use. Instead, they rely on the fact that within a given scope, there is only one resource with a given name, and can check using the names.

The other problem is the implications of multiple uses of modules within a single system. If a module is used more than once in a system, does it get instantiated each time, or is is the same module? The definitions of subsystem construction require each use to require resources, which may lead to more than one set of bindings. These issues are not discussed in any detail in most of the MILs.

We observed one facet in which our approach is not as intuitive as we would like. Several of the relations that appear in the core model have different characteristics depending on the particular extension. For example, the *matches* relation models the binding between a resource required by name and the actual resource. Ideally, *matches* should be a relation derived from the relations *provides, requires,* and *member of.* However, the theory model paradigm requires us to produce a core model that is consistent, and can be extended to produce the other models. Thus the *matches* relation is specified in all cases, and each extension provides different constraints. We would prefer to leave it as a derived relation, and specify different derived rules in each extension. Another example is that the *requires* relation in MIL 75 should be a derived relation from the *requires derived* and *requires nonderived* relations. But in other models, it is a primary relation.

It is worth noting that the standard theory is not sufficient to represent a generic MIL. The MILs that we have examined differ in the constraints for the *matches* relation, and on the constraints on the *provides* and *requires* relations. The minimum extensions needed to model any MIL are these two sets of constraints.

This paper has shown that the Theory-Model paradigm can be used to formalize and compare the structural representation of module interconnection languages. In performing the exercise, we have discovered several possible sources of confusion, and enhanced our understanding of each of the MILs.

References

[Chen 76]

Chen, P., “The Entity-Relationship Model: Towards a Unified View of Data”, *ACM Transactions on Database Systems*, Vol. 1, No. 1, 1976.

[Cooprider 79]

Cooprider, L., *The Representation of Families of Software Systems,* Ph. D. Thesis, Carnegie-Mellon University, Computer Science Department, 1979.

[DeRemer & Kron 76]

DeRemer, F., Kron, H., “Programming-in-the-Large Versus Programming-in-the-Small”, *IEEE*

Transactions on Software Engineering, Vol. 2, No. 2, June 1976.

[Holt *et al.* 88]

Holt, R. C., Matthews, P.A., Rosselet, J. A., Cordy, J. R., *The Turing Programming Language: Design and Definition*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Kerninghan & Richie]

Kernighan, B., Ritchie, D. M. *The C Programming Language*, Prentice–Hall, Englewood Cliffs, New Jersey, 1978.

[Lamb *et al.* 90]

Lamb, D. A., Jain, N., Ryman, A., *A Theory-Model Formalization of Jackson System Development*, Technical Report ISSN-0836-0227-90-269, Department of Computing and Information Science, Queen's University, published simultaneously as IBM TR-74.051,Oct. 1990.

[Liskov *et al.* 81]

Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J., Scheifler, R., Snyder, A., CLU Reference Manual, *Lecture Notes in Computer Science*, No. 114, Springer–Verlag, Berlin, 1981.

[Narayanaswamy & Scacchi 87a]

Narayanaswamy, K., Scacchi, W., "Maintaining Configurations of Evolving Software Systems", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 3, March 1987, pp. 324-334.

[Narayanaswamy &Scacchi 87b]

Narayanaswamy, K., Scacchi, W., "A Database Foundation to support Software System Evolution", *The Journal of Systems and Software*, Vol. 7, 1987, pp. 37-49.

[Parnas 72a]

Parnas, D. L., "A Technique for Software Module Specification with Examples", *CACM*, Vol. 15, No. 5, May 1972, pp. 330-336.

[Parnas 72b]

Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules", *CACM*, Vol 15, No. 12, December 1972, pp. 1053-1058.

[Prieto-Diaz & Neighbors 86]

Prieto-Diaz, R., Neighbors, J. M., "Module Interconnection Languages", *The Journal of Systems and Software*, Vol. 6, 1986.

[Rumbaugh *et al.* 91]

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object–Oriented Modeling and Design*, Prentice hall, Englewood Cliffs, New Jersey, 1991.

[Ryman 89]

Ryman, A., "The Theory-Model Paradigm in Software Design", IBM Tech Report No. TR74.048, IBM Toronto, Ont., Oct. 1989.

[Thomas 76]

Thomas, J., Module Interconnection in Programming Systems supporting Abstraction, Phesis, Brown University, Providence, R. I., April 1976.

[Tichy 80]

Tichy, W., Software Development Control Based on System Structure Description, Ph. D. Thesis, Carnegie-Mellon University, Computer Science Department, 1980.

[Wirth 71]

Wirth, N., "Program Development by Stepwise Refinement", *CACM*, Vol. 14, No. 4, April 1971, pp. 221-227.

[Wirth 85]

Wirth, N., *Programming in Modula-2*, third ed., Springer-Verlag, New York, 1985.

[Wulf 74]
Wulf, W. A., “ALPHARD: Toward a Language to Support Structured Programs,” Carnegie–Mellon Technical Report, April 1974.