Draco: A Method for Engineering Reusable Software Systems[1]

May 1, 1987

James M. Neighbors
Bayfront Technologies, Inc.

This work was published in final form as: Neighbors, J.M., "Draco: A Method for Engineering Reusable Software Systems", Chapter 12 of **Software Reusability**, *Volume 1: Concepts and Models*, Biggerstaff, T.J, and Perlis, A.J., eds., ACM Press Frontier Series, pp. 295-319, Addison-Wesley, 1989.

System Analysis, Design and Assessment the original author affliation was accquired by Bayfront Technologies, Inc. in 1995.

---

1. This work was supported by the National Science Foundation under grant MCS-81-03718 and by the Air Force Office of Scientific Research (AFOSR).

Contents

1 Introduction

Everyone is looking for an order of magnitude increase in the production of software systems; but, historically, such increases have never been achieved. Certainly such an increase will not be the result of simple extensions of current techniques. Many factors have contributed to the current *"software crisis"*.

- The price/performance ratio of computing hardware has been decreasing about 20% per year [Morrissey79].
- The total installed processing capacity is increasing at better than 40% per year [Morrissey79].
- As computers become less expensive, they are used in more application areas all of which demand software.
- The cost of software as a percentage cost of a total computing system has been steadily increasing. The cost of hardware as a percentage cost of a total computing system has been steadily decreasing [Boehm81].
- The productivity of the software creation process has increased only 3%-8% per year for the last thirty years [Morrissey79]. This increase in productivity includes all the developments in software engineering and the development of higher-level languages.
- There is a shortage of qualified personnel to create software [Lentz80].
- As the size of a software system grows, it becomes increasingly hard to construct.

The "software crisis" is not a problem of small systems. Adequate methods exist for a single programmer to produce 10k lines of high-level source code or five programmers to produce 50k lines of high-level source. Perhaps finding people who are familiar with the development techniques is difficult, but the methods appear adequate. Software development becomes a crisis when twenty people attempt to cooperate in the development of a 200k line system. Systems of this size have murky and ambiguous specifications. The social interactions of the developing team members become a major expense of time.

The interest in *reusable software* stems from the realization that one way to increase productivity during the production of a particular system is to produce less software for that system while achieving the same fuctionality. This can be done by building the system out of *reusable software components* and amortizing the cost of developing the general software components over the construction costs of many systems.

The Draco approach to the construction of software from reusable software components described in this paper neither deals with the important problems of organizational interactions of developing team members nor methods for the complete specification of software systems. Instead we focus only on the constructive aspects of software production (analysis, design, implementation) under the assumption that with such an approach the number of development team members producing a large system could be drastically cut and the specification clarified using a rapid development feedback cycle with the original specifiers.

The first Draco prototype was completed in 1979[Neighbors80, Neighbors84b, Freeman87] and the last major revision of the mechanism was completed in 1983[Neighbors84a]. Since that time

the instrumental use of the mechanism has been stressed to understand its limits and pitfalls [Gonzalez81, Sundfor83a, Sundfor83b, Arango86]. This paper discusses the approach, including what we perceive as necessary future changes to the method to attempt the construction of truly large systems. These changes have not been implemented and experimented with on real systems.

2 Methods of Software Reuse

Before we discuss the Draco approach to the problem of software reuse it will be useful to characterize the three basic approaches to the problem. These are extreme points of view along which different approaches can be characterized. Of course, all approaches contain some aspects of each view.

2.1 Libraries of Reusable Components

The most obvious approach to the problem of software reuse is to form libraries of software modules; but when we consider the reuse of existing programs we must be careful in describing the goals of the reuse. In some cases a programmer is looking for a program part which can just be "plugged in" without modification. In other cases the programmer is looking for a program part which can be modified before use. This is an important consideration in the design of a library of reusable program parts. In the first instance only what the program part does need be stored while in the second instance both what the part does and how it does it need be stored.

The reuse of program parts *without any modification* is extremely successful. The obvious example of this approach is the implementation of compilers by linkage to run-time support routines. The use of classical program libraries supported by linkage editors is another example of reuse of this type. This reuse is invisible to the programmers requesting the reuse. The library is kept in an encoded form and thus cannot be changed by the programmer. This approach has been most successful in reusing libraries of mathematical functions where the data objects being manipulated are one of a few different types of number representation.

The reuse of program parts *modified with the aid of a machine* has not been investigated very much as such. Most of the work in this area is from research on automatic programming, program generators, computer-aided software engineering, and specialized language design. The reuse of program parts *modified without the aid of a machine* is a major activity of detailed design and coding. Encyclopedic works such as [Knuth68, Sedgewick84, Press86] serve as guides supplying information *above the level of programming language code* which tells the programmer what the part does and how it does it. This "how" information allows the programmer to adapt the part to the system under consideration.

2.1.1 Problems with Software Part Libraries

If we have the "what" describing the function of each software part in the collection, then one straightforward way of organizing the collection is to put each part into a library of source code. Potential users of the part would search through the "what" descriptions of the parts of the library and select the appropriate part. This is the scheme used by most source program libraries. The problems encountered by this scheme are:

1. *classification problem*: What is an appropriate language or scheme for specifying and searching "what" descriptions?

2. *search problem*: The burden of searching the library is placed on the potential user of a part. Quite often it is easier for a potential user to (re-)build a part from scratch rather than find a part in a library and understand the constraints on its use and the ramifications of its design decisions.

In addition, following the previous discussion, if the potential user is looking through the library for a software part which can be modified, software part libraries will encounter the following problems:

1. *structural specification problem*: What is an appropriate language or scheme for specifying "how" descriptions and constraints of usage between software parts?

2. *flexibility problem*: Which design and implementation decisions are flexible and which are fixed in each of the software parts in the library.

2.1.2 The Overall Library Problem

The overall *library problem* is aggravated by and increases the magnitude of all the other problems. If the parts in the library are to be modified and reused then they must be small to be general, flexible, and understandable. However, if the parts in the library are small then the number of parts in a usable library must be very large. These two objectives are always in conflict. If a library contains many small parts then it lessens the structural specification and flexibility problems at the expense of increasing the classification and searching problems. If a library contains a small number of large parts then it lessens the classification and searching problems at the expense of increasing the structural specification and flexibility problems. Some interesting work dealing with these issues has been done[PrietoDiaz87].

2.1.3 Specialized Languages

An alternative to program libraries is to use specialized languages as surface forms to tie together software parts. As an historical example consider FORTRAN not as a programming language but as a surface description scheme for tying together the software parts which make up the FORTRAN run-time library. Would FORTRAN have been nearly as successful if it had been presented as a "library of interesting and useful numeric input, calculation, and output routines with descriptions"? Just a library would not have been as successful because the burden of using the library is placed upon each and every potential user of the library. Having a surface language which ties the library together in restricted ways removes this burden. Fahlman's NETL[Fahlman79], the CCITT protocol description language[CCITT84], and Mallgren's specification of graphics languages[Mallgren83] are all recent examples of this technique in problem domains far removed from FORTRAN. These are problem domain specific languages.

2.2 Narrow Spectrum Transformational Schemes

In a narrow spectrum transformational approach a system description is refined through a discrete series of *narrow spectrum languages*. In the refinement of the system it is held in only one language at a time and the system goes through stepwise refinement from one language to the next. Each discrete level of language has its own modes of analysis and model of completeness. *The languages for describing the "waterfall" software engineering cycle are narrow spectrum languages.* Each is concerned with a different aspect of the developing system. The following is a general description of the narrow spectrum languages found to be useful when following the software engineering "waterfall" lifecycle model.

- **Requirements** languages capture the external environment in which the system under consideration must work and the required external operation of the system. It is the interface specification of the proposed system with the rest of the world. Most system requirements are captured in natural language.

- **Analysis** languages capture the answer to the questions "What functions are required within the system?" and "What information is produced and consumed by each function?". This information is usually captured in the form of graphical data flow diagrams (DFDs)[Ross77, Gane79].

- **Architectural Design** languages focus on capturing the control flow of the developing system to answer the questions "Which of these functions are tightly coupled in data and control flow?". The goal here is to minimize coupling and maximize cohesion. This information is usually captured as graphical control flow hierarchy trees with data passing and sharing annotations [Yourdon79, Jackson76]. The result of this partitioning is a collection of tightly coupled functions and procedures called a module.

- **Detailed Design or Implementation** languages focus on the control flow within an individual function or procedure, composite data definitions, and the definition of its interface with the rest of the system. The form is usually a pseudocode of control flow constructs and function or procedure passed parameter declarations supported by a data dictionary [Yourdon79, Caine75].

The languages described above are very general in their form and apply to all kinds of systems[2]. *There is no direct relation between the use of the narrow spectrum transformational approach and a narrow generality of systems produced*. However, these general ideas can be tailored to a specific problem domain. Jackson Design is an example of such tailoring. Jackson Design[Jackson76] is a general model of the process of processing input forms, interacting with a database and producing output reports. The general notations specified above are restricted in Jackson Design to producing systems of this form.

Fourth Generation Languages(4GLs) are simply program generators which carry this tailoring process one step further. The process is so domain dependent that the translation between the narrow spectrum languages can be carried out by a mechanical agent. However, 4GLs still go through the same narrow spectrum transformational approach concerned with the same notations as the more general process outlined above.

2.2.1 Problems with the Narrow Spectrum Transformational Approach

Since the narrow spectrum approach captures a view of a developing system above the level of program code, there is some hope that these analysis and design models could be reused. If these models are formed into a library, these libraries will inherit all of the library problems mentioned in the previous section. The more tailored versions of the narrow spectrum approach, such as 4GLs, are an example of the daily reuse of analysis and design.

The two basic problems specific to the narrow spectrum approach are:

- How do we make the jump from the narrow spectrum language which currently describes the system to the next abstraction level down?

---

2. With the possible exception of real-time systems which requires time constraints to be added to each of the language levels.

- Once we have made the jump from one abstraction level to the next, what do we do if we discover that our work in the previous abstraction level was incomplete? Can we backup without undoing all of the work we did to get here?

The top-down rigidity of the software engineering "waterfall" model of system development is aa aspect of the second problem. Early proponents of the "waterfall" model did not intend it as a strictly top-down process without backup[Royce70]; but, the difficulties developers had in dealing with the second problem caused it to evolve into such a process.

2.3 Wide-Spectrum Transformational Schemes

A wide-spectrum transformational approach uses one *wide spectrum language* to describe the developing system from its requirements to its final implementation[3] level. The requirements statement is "transformed" (i.e., refined) into lower level constructs nearer to implementation. At any one time the wide-spectrum statement of the system being refined will include statements from all of the modeling phases correspondent in the narrow spectrum transformational approach (e.g., the statement could contain data flow constraints as well as control flow constraints). Thus, the wide-spectrum statement refined to the implementation level contains the complete *refinement history* of the process. The wide-spectrum transformational approach is the predominate approach in knowledge based automatic programming[Balzer81, Cheatham84, Smith85, Waters85, Green76].

2.3.1 Problems with the Wide-Spectrum Transformational Approach

The wide-spectrum language must of necessity span quite a range of description from the model of the external world in the requirements to the description of indivisible data item operations in the implementation. The problem of using wide-spectrum languages for requirements is similar to the problem of using very formal wide-spectrum languages (e.g., 2nd order predicate calculus) for requirements. Prospective users of a wide-spectrum approach should be concerned with the following questions:

- How is knowledge about the world encapsulated for reuse using the wide-spectrum language primitives so that we don't end up describing standard high-level constructs like physical matter or low-level constructs like priority queues over and over again during the refinement of many requirements statements?

- What encapsulations come already provided?

- Can I change these encapsulations if they do not meet my needs?

- How can I be assured that the language will not become bloated and thus too complicated to learn as new constructs are perceived to be needed on the many levels of abstraction and in the many problem domains which the language must represent?

In order to capitalize on reuse, wide-spectrum approaches must provide a mechanism where the general wide spectrum language can be tailored to certain problem areas. Otherwise there is nothing to reuse and system descriptions must be stated in terms of first principles each time. The work of providing problem domain specific customizations of wide-spectrum languages is underway[Barstow85, Wile86].

---

3. The final implementation description only contains the control and data description constructs of a conventional high-level language which could be compiled by a conventional compiler.

2.4 Summary

As stated earlier the viewpoints represented above are extreme and no system takes entirely one view. The Draco approach described in detail in the next section borrows from each of the viewpoints. In particular, Draco uses a library of problem domain-specific notations, each of which is narrow spectrum in scope. These are not arranged in a strict hierarchy for step-wise translation as in the narrow spectrum approach. Instead, a single mechanism which spans the complete wide-spectrum range of abstraction manages refinement using the knowledge specified in all the known domains.

3 The Draco Approach

The Draco approach to the construction of software systems from reusable component parts is strongly influenced by our viewpoint as practicing software engineers. The basic idea captures the frustrating feeling that most of the system you are currently building is the same as the last few systems you have built; but once again you are building everything from scratch. The current system development is behind schedule and you have no time to figure out what this similarity means.

3.1 Purpose and Viewpoint

Our point of view in the analysis and design of Draco is an engineering point of view. We are not trying to advance the state of the art in knowledge representation, language design, parser generation, module interconnection languages, program transformations, or planning. Instead we are attempting to discern which techniques have been successful in these areas, fuse them into an experimental system, and see where the system fails.

As software engineers we are concerned with how Draco would be used by an organization engineering large, real systems. We are attempting to address the "software crisis" as described above. This is not a crisis in building small systems[4] but a crisis in building large systems.

3.2 Organizational Use of Draco

Figure 1 shows the flow of information between people in different roles external to Draco.

4. Although historically devices such as structure programming developed for use in large systems tend to aid the development of small systems.

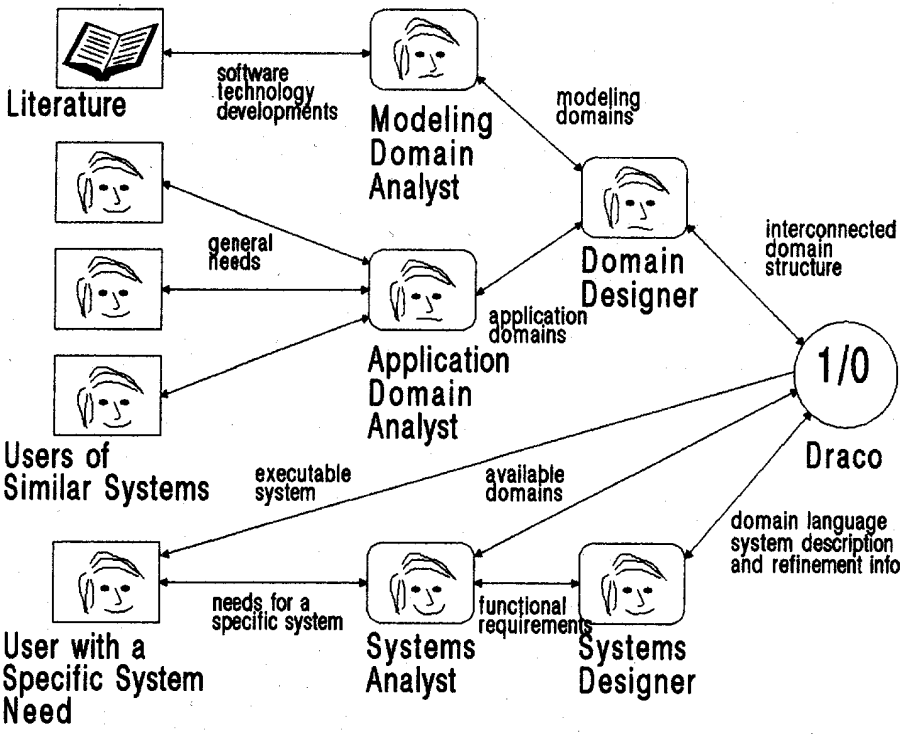

Figure 1. Organizational Context of Draco

Classically, during the system analysis phase of software construction a user with a desire for a certain type of system would interact with a systems analyst who would specify *what* the system should do based on the analyst's past experience with these types of systems. This would be passed on to system designers who would specify *how* the system was to perform its function.

With Draco we hypothesize three new major human roles: the application domain analyst, the modeling domain analyst, and the domain designer. An *application domain analyst* is a person who examines the needs and requirements of a collection of systems which seem "similar". We have found that this work is only successfully done by a person who has built many systems for different clients in the same problem area. We refer to the encapsulation of this problem area as a *domain*. Once the domain analyst has described the objects and operations which are germane to an area of interest then these are given to a *domain designer* who specifies different implementations for these objects and operations *in terms of the other domains already known to Draco*. The *modeling domain analyst* performs a function similar to the application domain analyst, but is more concerned with which notations and techniques have been successful in modeling a wide range of applications. The particular information needed to specify a domain is given in the following section.

Once a set of Draco domains has been developed by an organization in their area of software system construction, then new system requirements from users can be considered by the organization's *systems analysts* in the light of the Draco domains which already exist. If a Draco domain exists which can acceptably describe the objects and operations of a new system, then the systems analyst has a framework on which to hang the new specification. This is the *reuse of analysis information* and in our opinion it is the most powerful brand of reuse. Once the new system is cast as a domain language notation then the *systems designer* interacts with Draco in the refinement of the problem to executable code. In this interaction the systems designer has the ability to make decisions between different implementations as specified by the domain

designers of the Draco domains. This is the *reuse of design information* and it is the second most powerful brand of reuse.

Thus, Draco captures the experience of the "old hands" of the organization and delivers this experience in problem specific terms to *every* systems analyst in the organization for their education and use.

3.3 Architectural Design of the Draco Approach

From the above discussion of using Draco within an organization it is clear that there are three basic points of concern to the different users of the approach: the individual domains, the interrelationships between the existing domains (i.e., the *domain structure*), and how the Draco mechanism controls the refinement of a particular system. We shall deal with these individual points in succeeding sections.

4 What Comprises a Domain Description

In this section we will describe the results of domain analysis and domain design which must be given to the Draco mechanism to specify a complete domain. There are six parts to a domain description:

4.1 Parser

The parser description defines the interface between the domain and the mechanism. There are three parts to the parser description:

1. The **external syntax** of the domain and the *internal form* of the domain is described in a conventional BNF notation which is augmented with control mechanisms such as parser error recovery and parser backtracking. The internal form is a tree with an attribute name and data at each node. The internal form is the data actually manipulated by the Draco mechanism.

2. The parser description must define what is a well formed formula in the domain's internal form. This is a **semantic check** on the combination of objects and operations in the domain. This subsumes a check the production and consumption of data by the domain.

3. Finally, the parser description must specify the **database schema** for the information to be maintained by the mechanism for the exclusive use of the agents of the domain.

As Draco manipulates the internal form of a domain, the parser description is the final arbiter of what constitutes a valid notation in the domain both as a fragment of notation and as a complete notation statement. This information can be used to prohibit or trigger the use of other domain definitions by the mechanism. As an example, the refinement of an operation component in the domain may be held up until the semantic checker is convinced that the objects input to the operation are semantically valid types.

4.2 Prettyprinter

The prettyprinter description tells Draco how to produce the external syntax of the domain for all possible *notation fragments* in the internal form of the domain. This is necessary for the

mechanism to be able to interact with users in the language of the domain and discuss incomplete parts of the developing system.

Since the prettyprinter is the only agent of the domain which can communicate with the systems designer, it must also be able to present the information gained from the other domain-specific agents described below.

4.3 Optimizations

The optimizations[5] represent the rules of exchange between the objects and operations of the domain. Optimizations only work within the domain from which they were specified. They never cross domain boundaries. There are three parts to the optimization specifications:

1. **Source-to-source optimizing rules** are simple source pattern to source pattern rewrite rules similar to the source-to-source program transformation work[Kibler77].

2. **Source-to-source optimizing procedures** are procedures which may or may not be triggered by a source pattern which take an instance of the domains internal form as an argument and produces a new instance of the domains internal form.

3. **Optimization application scripts** describe possible structured interactions[6] developed by the domain designer which the optimizing rules and procedures can provide to the system designer. These can also be used as an element in refinement planning by the refinement mechanism.

The output domain language fragment of all of the optimizers is subject to the scrutiny of the parser description as the final arbiter of a well formed notation fragment in the domain. The semantic equivalence of the optimized result is not checked. *The optimizations are guaranteed to be correct independent of any particular implementation (i.e., component refinement) chosen for any object or operation in the domain*. This ganularity of meaning is important and we will see later how it provides us with powerful domain dependent optimizations.

4.4 Components

The *software components* specify the semantics of the domain. There is one software component for each object and operation in the domain. The software components make *implementation decisions*. Each software component consists of one or more *refinements* which represent the different implementations for the object or operation. *Each refinement is a restatement of the semantics of the object or operation in terms of one or more domain languages known to Draco*. Thus component refinements cross domain boundaries. Conceptually, it is easiest to view each refinement as a macro body[7] for the domain object or operation it represents. The macro body is written in terms of other (perhaps the same but not usually) domain notations.

---

5. Previously these were referred to as transformations in the source-to-source transformation sense. However, since the wide spectrum approaches refer to transformations as operations which make implementation decisions (i.e., refinement decisions) we decided on a more appropriate name.

6. Previously these were specific to the mechanism and were called tactics

7. Simple macro expansion or instantiation inline with suitable systematic renaming is only one possible alternative for architectural design. The applicative feel of the process, however, makes it a comfortable model.

Components are the only part of a domain description which cross domain boundaries (i.e., components need to know about other domains) and these are patterns they are **not** procedures. We do not use procedures for this function since the wide-spectrum mechanism must be able to analyze the possible inter-domain connections made by a component.

4.5 Generators

Generators are domain-specific procedures which are used in circumstances where the knowledge to do domain-specific code generation is algorithmic in nature. This is analogous to program generators. The procedure is not doing an optimization task but actually writing new code in the domain. The construction of LR(k) parser tables from a grammar description and the normalization of database schemas are two examples of generators. As with optimizing procedures, generators only operate and produce the internal form of the one domain where they are described. The resulting output notation fragment is checked by the parser description.

4.6 Analyzers

Analyzers are domain-specific procedures which gather information about an input instance of domain notation. As with all other procedural specifications in a domain definition, a particular analyzer only works with the specific domain where it was defined. As with all domain-specific procedures, the data produced and consumed by each analyzer is kept under the schema described in the domain parser definition. The actual data is managed by the Draco mechanism which is described in a later section. Dataflow analyzers, execution monitors, theorem provers, and design quality measures are examples of analyzers.

4.7 Domain Description Summary

Thus, the basis of the Draco work is the use of *domain analysis* to produce *domain languages*. Once a statement in a domain language has been *parsed* into internal form it may be:

1. *Prettyprinted* back into the external syntax of the domain.
2. *Optimized* into a statement in the same domain language.
3. Taken as input to a program *generator* which restates the problem in the same domain.
4. *Analyzed* for possible leads for optimization, generation, or refinement.
5. Implemented by *software components* each of which contains multiple *refinements* which make implementation decisions by restating the problem in other domain languages.

5 The Nature and Structure of Domains

Since the semantics of the components of one domain are described by translation into the components of other domains, a hierarchy of domains is formed . The structure of domains thus formed is a cyclic[8] directed graph. Obviously, if we ever expect to produce executable systems, some of the domains are the equivalent of some conventional programming language. These are called *executable domains*. We define the *level of abstraction of a domain* with respect to an

8. The graph is cyclic because in many case two modeling domains will use each other as modeling domains.

executable domain to be the longest acyclic path from any of the refinements of the domain's components to the executable domain. The domains with the highest levels of abstraction are called *application domains* while the domains in the abstraction levels between the application domains and execution domains are called *modeling domains*.

The idea of a hierarchy of domains came from early work on source-to-source program transformations[Standish76, Kibler77]. In that work we were attempting to perform the *specialization* of a general high-level language program in its source form so that the programmer could see the modification. As an example, we would refine a general Pascal matrix multiply program under the specialization that one of the matrices was an upper triangular matrix. We would chain together many transformations, some specializations and some general compiler-like optimizations, to produce the resulting program. Consider the case of matrix multiplication with the identity matrix. In Pascal the source-to-source transformation system had to work very hard to realize that a matrix multiply becomes a matrix copy. In APL it is a single, trivial transformation. APL performs matrix operations at a *higher level of abstraction* than Pascal. Clearly, it is important to perform optimizations at the highest level of abstraction possible. We explored domain-specific high-level languages and discovered that not only were significant optimizations easier in these languages; but systems specification and synthesis were also easier since they were free of low-level implementation details.

5.1 Application Domains

There are two kinds of *domain analysts*: those that primarily construct modeling domains and those that primarily construct application domains. A database is not a complete application but many applications use its services. It comes as no surprise that an accounting systems expert is not a database expert; but most accounting systems use a database.

From our experience, application domains become a kind of "glue" which ties existing modeling domains together in a restricted way. An accounting systems application domain would not allow full access to all of a database domain's capabilities. Instead the accounting systems domain would form a model of accounting objects and operations in the notation of the database domain. Accounting objects like journals and ledgers are each a restricted class of general database objects like relations. All journals and ledgers in all systems constructed using a particular accounting systems domain have a certain basic form. Of course, this basic form can be expanded in different ways in constructing different systems; but the core descriptions of the accounting objects never change. Similarly, an accounting operation like posting is a specific type of restricted database operation.

An application domain analyst must be able to view all of the possible modeling domains as relatively simple data flow processes without worrying about all the details. As an example, one model of a database is a process which when given a schema, a query or fact, and a database produces a relation. A relation is a set of records which can be generated one at a time in a specified order. This is a very simple description of a very complex mechanism. It is not clear what is done during refinement and what is done during execution. The application domain analyst is really trying to do a data flow analysis model for one particular, familiar domain using the basic building blocks of the modeling domains.

5.2 Modeling Domains

Most of the domains known to Draco will be modeling domains. A modeling domain is **not** a complete application but the encapsulation of the engineering knowledge necessary to produce a significant, but well-defined subpart of a complete application[Rowe78]. The concept of many domains sets the Draco approach apart from other approaches to software reuse. *A modeling*

domain creates partitions in the knowledge needed to construct systems, similar to the way a module creates a partition in the control and data flow of the system itself. A system module is a collection of functions, procedures, private definitions, and public definitions which performs a significant, encapsulated function for the system as a whole. Analogously, a modeling domain is a collection of objects, operations, optimizations, and semantic translations which encapsulates the analysis, designs, and implementations of software parts which perform the major function represented by the domain. This does not mean that the refinement of a domain will necessarily map into a module in the final system. As Balzer[Balzer81] has described, "the process of refinement is the spreading of information through the developing system". Initially, however, this information must be in one place.

In the preceeding description of a domain structure notice that we have scrupulously avoided the refinement of one domain into another by a procedure. This is by design since it couples the domains too tightly and makes the knowledge base hard to analyze[9]. The mechanism which manages the refinement must be able to analyze the refinement process. Similarly, programmers which manage the control and data flow in software systems must have modules in order to be able to analyze the control and data flow process.

In our experiments using the Draco method, understanding what is an appropriate modeling domain seems to have been the hardest problem. This is not a surprise. Doing the module decomposition of a single concrete system is hard enough. The partitioning and selection of domains in Draco is not driven by a single specification but by the domain analysts experience in the construction of many systems. The best training for a domain analyst is to know computer science and participate in the construction of many types of systems.

5.3 Execution Domains

In the beginning we assumed that there could be multiple executable domains, and, indeed, there can; but, building execution domains represents a pitfall for computer scientists. Computer scientists are well trained in general control flow and data structure constructs. Thus, they are domain experts in this area. Naturally, domain experts like to build and experiment with domains in their area of expertise. The explosive growth in the number of general-purpose languages in the 1960s bears witness to this fact. In our early work we fell victim to this desire[10]. We now believe that the selection of one executable domain, a *base language*, is the key to keeping the focus on truly domain-specific languages.

Once a base language has been selected, the compiling can be left to the compilers and the truly powerful application-specific domains can be explored. This does not mean that Draco could not be used for the production of a compiler, it is just that the construction of compilers is not the problem precipitating the "software crisis". In our experience we have learned that the following features are required of the base language.

- **function/procedure parameter passing model**: As the mechanism keeps track of the system being refined many architectural design models are possible. A refinement may be instantiated inline or a procedure may be created to save space. Without a model of the process of making a procedure or function and passing information to it, the mechanism becomes very difficult. Consider attempting to handle Lisp's EXPRS,

9. An analyzing system would have to understand the refining procedures and program understanding even in simple domains has not been successful.

10. Some constructed domains: MC68000 assembly, Intel8080 assembly, Dec20 assembly, Lisp primitives, stack machine assembly, SIMAL: an Algol-like language, RML: a Pascal-like language with modules.

FEXPRS, LEXPRS; Pascal's call by value unless its a structure; and FORTRAN's ancient call by return value.

- **module, module interconnection and scoping model**: As the mechanism constructs new parts of the system it sometimes needs to create names to call things and only enables certain other parts of the system to access them. Consider attempting to handle Lisp's dynamic scoping and Pascal's lexical scoping. In addition, the mechanism needs the concepts of module and module interconnection for much the same reason that people do, to bound the context of definitions in larger systems.

- **parallel processing model**: Many concepts, such as natural language parsing using an augmented transition network (ATN) are naturally described as parallel processes. In fact its quite a bit of work to remove the parallelism. It should be possible to describe parallel processes in a parallel form with the chance of a multiprocessing computer being able to directly exploit the resulting parallel system.

- **exception handling model**: All of the refinements are based upon the principles of abstract data types and each different implementation of the objects and operations have different exceptions which can occur. Unless there is exception handling in the base language, the Draco approach will produce systems which spend time explicitly looking for exceptions which may have occurred. Exceptions should happen occasionally and indicate a interruption in the main control flow processing. The main control flow processing should not be checking for them explicitly all the time.

- **standard control flow and data structure definitions**: These are of interest if the resulting system is to ever be understood by a regular programmer.

- **compatible compiler implementation available**: Once something has been refined it should be able to be executed on a collection of machines.

The base language represents a high-level model of the computer on which we expect the resulting system to run. The above description is for a von Neumann computer. I would not expect the base language for a massively parallel computer to be the same.

For all of these considerations, we have chosen Ada[USDODAda83] as our base language for future work. In addition, we hope to do future mechanism work in Ada. *The choice of a base language is **not** important as long as one is chosen and it meets the above criteria.* The construction of large, reliable, application systems in the base language is the problem; not the development of another general-purpose, high-level language.

6 The Draco Mechanism

The Draco mechanism is the part which interacts with all of the human roles. It must provide support to application domain analysts, modeling domain analysts, and domain designers in their efforts to add to the knowledge base. It must provide support to systems analysts and system designers in their respective functions of specifying and refining a particular system. It maintains the wide-spectrum model of the developing system.

6.1 The Basic Refinement Cycle

Once a system description has been cast in the notation of an application domain by a system analyst, then the system designer uses the basic cycle of selecting a set of instances of an application or modeling domain in the developing refinement and restricting the refinement focus to only the instances selected. Within these selected instances the system designer would use the domain's optimizers, generators and analyzers before deciding whether to use the domain's components to refine or not.

This *basic refinement cycle* produces the following view of domains during the refinement process. The system is originally described in a single application domain language, but the first refinement will introduce the notations of a modeling domain. Eventually, the developing system is described in the notations of many modeling domains at once. Figure 2 graphically illustrates the refinement process from a statement in only the application domain (NLRDB), through many modeling domains (ATN and RDB), and into the final executable domain (ADA).

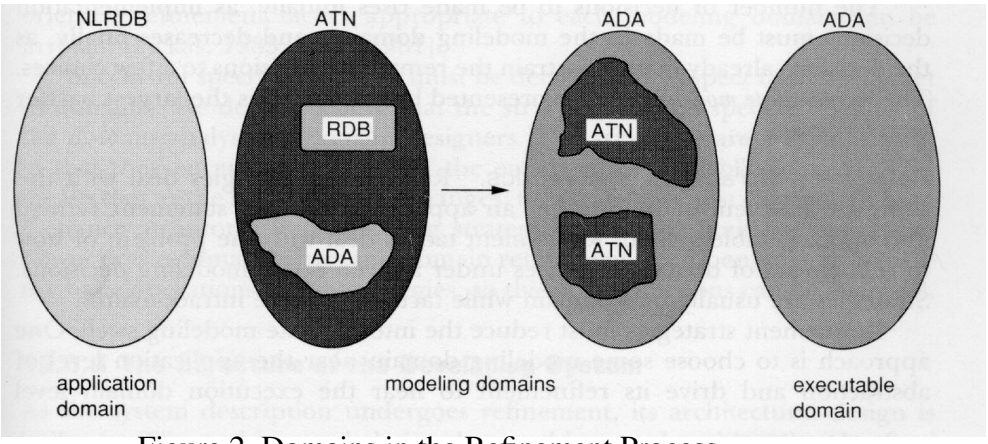

Figure 2. Domains in the Refinement Process

The systems designer works with the refinement mechanism in one domain at a time. In a developing system there probably will be multiple *instances* of a domain. The refinement mechanism may be directed to work with all instances during refinement or focus on a single one.

The concept of domain here is very useful in supplying a psychological set to the systems designer (i.e., the designer must only consider and think about the objects and operations of one domain at a time.) The ability to provide a psychological set to the systems designer is lost if the underlying representation of the developing system is a wide-spectrum language. This applies even in a full *automatic programming system* where the system designer is an automated agent. The selection of domains by the system designer for refinement provides a method of *progressively deepening*[Simon69] the system description during the refinement process.

6.2 Managing the Refinement Process

As with the refinement of systems by conventional means, the refinement process does **not** proceed strictly top-down from one modeling domain to another or from modeling domains at one level of abstraction to modeling domains at a lower level of abstraction. Sometimes it is necessary to backup the refinement process to remove an overly restrictive decision. As the process proceeds, a *refinement history* is recorded which can supply a top-down derivation for each statement in the resulting executable system. The refinement history tends to be much larger than the resulting program code[11]. There are two uses for this refinement history: to understand

11. Our best estimate is that the refinement history is about 10 times the size of the resulting source code.

the resulting system at different levels of abstraction and to guide the *refinement replay* of the problem if the original specification is changed and a new implementation is needed[Wile83].

In general we have found the making of design decisions to proceed as shown in figure 3.

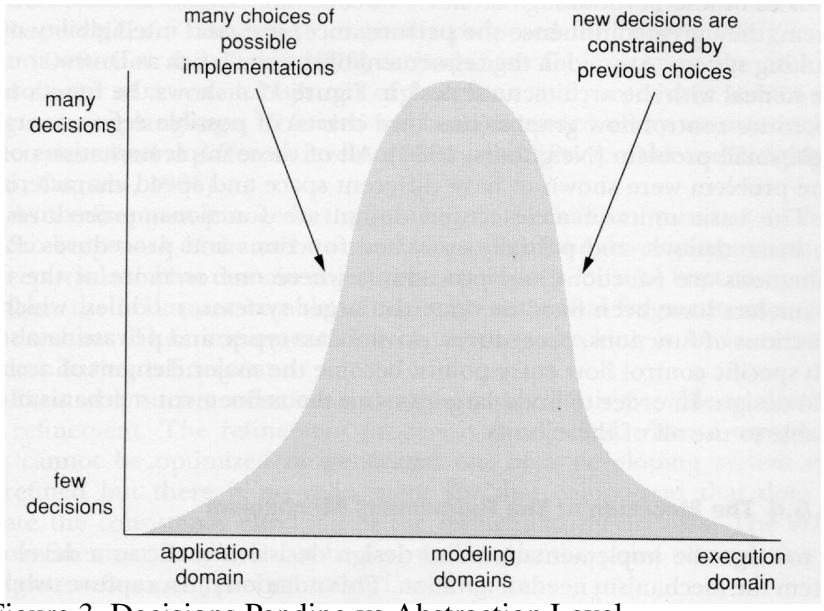

Figure 3. Decisions Pending vs Abstraction Level

The number of decisions to be made rises initially as implementation decisions must be made in the modeling domains and decreases finally as the already made decisions constrain the remaining decisions to a few choices. The *intermediate modeling swell* represented by this graph is the largest barrier to refinement.

6.2.1 Refinement Strategies and Tactics

Refinement strategies deal with the complete problem of how to get an application domain statement refined into an executable system. Refinement tactics deal with the problem of how to refine a set of domain instances under a given set of modeling decisions. Strategies are usually inter-domain while tactics are more intra-domain.

Refinement strategies must reduce the intermediate modeling swell. One approach is to choose some modeling domain near the application level of abstraction and drive its refinement to near the execution domain level of refinement. During this process many modeling decisions will be made and these modeling decisions will constrain the choices in other modeling domains.

Once some of the interfaces between modeling domains have been established by the early refinement of one part of the system, then the goal-oriented *refinement tactics* appropriate to each modeling domain can be invoked on the goal of meeting existing modeling decisions[Fickas85, Rich81].

Successful specific strategies must be derived with respect to a specific set of domains. We do not believe that the strategies can be specified by any of the domain analysts or domain designers. The strategies are wide-spectrum in that they encapsulate a view of the entire set of available domains. This means that as the set domains change, the strategies must change. To have a chance at automatically deriving strategies, we must *limit* the expressive power of the domain tactics and domain refinements (components) which are the basic operations of strategies so that their functions can be analyzed.

6.3 The Structure of the Developing System

As the system description undergoes refinement its architectural design is built up. The architectural design has nothing to do with what the final system does. In building a specific system there are many ways to partition the control flow into functions and procedures. Similarly, there are many ways to partition the data into composite data objects.

While these partitionings do not influence the semantics of the resulting system, they severely influence the performance, size and intelligibility of the resulting system. Any aid in the refinement of systems, such as Draco, must be able to deal with the issue of architectural design. Figure 4 shows the function and procedure control flow graphs (structure charts) of possible refinements of a single small problem[Neighbors80] refined by Draco.

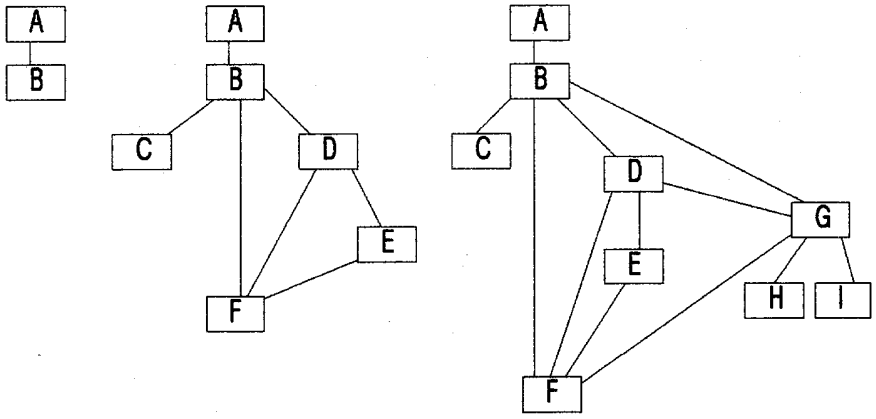

Figure 4. Different Architectural Designs Resulting from the Same Problem“

All of these implementations of the same problem were shown to have different space and speed characteristics. The basic units of architectural design are functions, procedures, inline instantiations, and partially evaluated functions and procedures. Partial evaluations are functions and procedures where one or more of the usual parameters have been fixed in value. In larger systems, modules which are collections of functions, procedures, private data types and private data stores with specific control flow entry points become the major element of architectural design. In order to build large systems the refinement mechanism must be able to construct architectural designs using all of these units.

6.4 The Notation of the Refinement Mechanism

To manage the implementation and design decisions made in a developing system the mechanism needs a notation. This notation must capture two kinds of information specific to the particular system being refined: implementation decisions of the different domain parts (detailed design and coding information) and the control and data flow structure of the developing system (architectural design information). In addition, the mechanism also needs a notation for describing and reasoning with the elements of the knowledge base represented by the complete set of described domains (domain analysis and design information).

A useful model for these notations has been *module interconnection languages*(MILs). These were originally proposed for *programming-in-the-large*[DeRemer76] as a language for capturing the architectural design of a system built out of functions and procedures written in a programming language. The idea has been extended many ways to include objects described at different levels of abstraction[PrietoDiaz86, Goguen86].

As a refinement proceeds, the refinement mechanism must use a kind of MIL to keep track of the architectural structure of the developing system and what domain objects and operations must be kept compatible. Further, the MIL must be able to deal with *notational fragments* of each of the domains. Thus, *implementation consistency checking* is an incremental process during refinement.

Even with such a notation correctly managed it is possible to *deadlock the refinement*. The refinement process is deadlocked when a component which cannot be optimized or generated out of a developing system must be refined and there is no refinement for that component which does not violate the consistency checking of the refinement mechanism. The effects of a refinement deadlock can be far-reaching to the point of requiring the complete refinement process to start over again.

6.5 SubSystems as Major Parts

One of the problems with the original implementation of Draco[Neighbors84a] was that the method required each systems designer to refine every statement in a developing system all the way down to the executable domain each time. Even though the system could aid the refinement through the use of tactics so that many tedious decisions did not need to be made, the process was still tedious even for the small (2k-4k line) programs produced. Sundfor was first to notice that large systems would really not be built this way[Sundfor83b] and our recent experience with understanding the structure of large systems enforces this belief.

The refinement mechanism must be able to use pre-refined, large subsystems as part of its reasoning process. These subsystems are nothing more than general implementations of commonly used modeling domains such as database operations and menu operations. The difference between using subsystems and external pieces of program code is that *the subsystems were refined by the Draco mechanism and the modeling decisions made during the refinement are related to the domains known to the mechanism and these modeling decisions are retained.* Thus, it is the availability of the *refinement history* of a subsystem which enables the mechanism to reuse it in the development of other systems.

Notice that the use of a subsystem meshes quite well with the basic approach of refinement strategies. If a collection of compatible subsystems may be used early in the refinement of a particular system, then they may make implementation and modeling decisions which constrain the intermediate modeling swell to a large degree. Refining a large system using subsystems may be easier than refining a small one without subsystems.

Finally, notice that all systems refined by Draco are candidate subsystems because they all have refinement histories. Some, application-specific systems, however, are less likely candidates for reuse than others.

7 Experience With The Draco Approach

7.1 Reuse of code

The large amount of information in a refinement history is lost to someone attempting to reuse an existing piece of source code. To combine two existing pieces of executable source code this information must be recreated to ensure that any exchanged representations are consistent. For this reason we expect the reuse of existing executable source code will have a limited long-term

benefit where whole systems are built from reusable parts. In the short-term, highly controlled source code libraries will provide a quick productivity gain.

In our opinion, significant reuse only occurs when the analysis and designs of systems are built from reusable parts. Since analysis and design information is domain-specific, only system refinement aids which directly address the problem of domain-specific knowledge will have any significant reuse capability.

7.2 Efficiency of Systems Built from Reusable Parts

Systems constructed from reusable parts by the Draco method are not inefficient. The method is capable of refining systems with different implementations and different architectural designs (i.e., modular structures). Each of these have different time-space execution characteristics. In addition, the domain-specific optimizations provide a degree of optimization far above the well known optimizations of a general-purpose language compiler. In the state machine description of a communications protocol a domain-specific optimization may be able to remove or combine states. There is no analogous optimization in general-purpose compilers (i.e., an execution domain) because the information which enabled us to perform the optimization is no longer in the source code. Users attempting to reuse source code without a refinement history will find that their optimization options are limited.

7.3 The Problem of Domain Analysis

Domain analysis is knowledge engineering applied to computer science (modeling domains) and computer applications (application domains). The different types of organizations which deal with this knowledge on a daily basis have quite different views of domain analysis.

- **Academic organizations** rightfully view domain analysis as an *engineering* process. It is not a discovery process where completely new theories are tried out. Instead, it is the process of reviewing previous work and attempting to determine which techniques were successful and which were not. Academic organizations in computer science seem to prefer to work on new theories. However, some of the most successful academic work[12] is a fusion and formalization of successful techniques.

- **Production organizations** are forever caught in a cycle of being focussed on the current system under development. A domain analysis must be motivated on the basis of its cost being *amortized* over the costs of many systems. Further, if the domain analysis is performed by an actual domain expert (as it must be if there is to be any chance of success), then the organization risks failure in the development of the current system by removing the expert from the stream of production during the domain analysis.

- **Research organizations** are caught in the middle between a flood of new theories from computer science which are untried in practice and the highly filtered information from the streams of software production. To understand what really works in practice the researchers must build actual systems. However, if the researchers build an actual system, there is a chance that the organization will become a production and support organization.

A production quality refinement aid would give each of these types of organizations an incentive and framework for domain analysis. In the mean time, the informal process of domain analysis from each type of organization continues in the literature.

---

12. The work by Mallgren on graphics languages[Mallgren83] was a 1982 ACM Distinguished Dissertation.

7.4 Future

After the recent period of experimentation with the current Draco mechanism, we anticipate another period of new mechanism development. This paper represents the analysis of the new mechanism.

For experimentation and explanation purposes, it would be helpful if the mechanism was capable of completely refining itself. In the current implementation[Neighbors84a] only some of the mechanism (e.g., parsers, prettyprinters, tactics interpreter, refinement library builder) were constructed using the technique. For Draco to refine the Draco mechanism we must describe an application domain for the *class* of systems similar to Draco. It is our hope that with the new mechanism such a domain description will be possible.

References

[Arango86] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon, "TMM: Software Maintenance by Transformation," *IEEE Software*, pp. 27-39, May 1986.

[Balzer81] R. Balzer, "Transformational Implementation: An Example" *IEEE Transactions on Software Engineering*, vol. 7, pp. 3-14, January 1981.

[Barstow85] D. Barstow, "Domain-specific Automatic Programming,", *IEEE Transactions on Software Engineering*, vol. 11, pp. 1321-1336, November 1985.

[Boehm81] B. Boehm, *Software Engineering Economics*, Prentice-Hall 1981.

[Caine75] S. Caine and E.K. Gordon, "PDL-A Tool for Software Design," in *Proceedings, National Computer Conference*, vol. 44, pp. 271-276, AFIPS Press 1975.

[CCITT84] CCITT, "Formal Description Techniques for Data Communications, Protocols, and Services," *CCITT Recommendation X.250*, 1984.

[Cheatham84] T.E. Cheatham "Reusability Through Program Transformation," *IEEE Transactions on Software Engineering*, vol. 10, pp. 589-594, September 1984.

[DeRemer76] F. DeRemer and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, vol. 2, pp. 80-86, June 1976.

[Fahlman79] S. Fahlman, *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press 1979.

[Fickas85] S. Fickas "Automating the Transformational Development of Software," *IEEE Transactions on Software Engineering* SE-11, (11), pp. 1268-1277, November 1985.

[Freeman83] P. Freeman, "Reusable Software Engineering: Concepts and Research Directions" in *Proceedings of the ITT Workshop on Reusability in Programming*, ITT, pp. 2-16, September 1983.

[Freeman87] P. Freeman, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," *IEEE Transactions on Software Engineerng*, in press 1987.

[Gane79] C. Gane and T. Sarson, *Structured Systems Analysis: tools and techniques*, Prentice-Hall, 1979.

[Green76] C. Green, "The Design of the PSI Program Synthesis System," in *2nd International Conference on Software Engineering*, pp. 4-18, October 1976.

[Goguen86] J. Goguen, "Reusing and Interconnecting Software Components," *IEEE Computer*, pp. 16-28, February 1986.

[Gonzalez81] L. Gonzalez, *A Domain Language for Processing Standardized Tests* (MS Thesis), University of California, Irvine, ICS Dept., 1981.

[Jackson76] M.A. Jackson, "Constructive Methods of Program Design," in *Proceedings, 1st Conference of the European Cooperation in Informatics*, vol 44. Springer-Verlag 1976.

[Knuth68] D. Knuth, *The Art of Computer Programming*, volumes 1-3, Addison-Wesley, 1968-1973.

[Kibler77] D. Kibler, J.M. Neighbors, and T.A. Standish, "Program Manipulation via an Efficient Production System", *SIGPLAN Notices*, vol. 12, no. 8, pp. 163-173, 1977.

[Lentz80] B. Lientz and E. Swanson, *Software Maintenance Management*, Addison-Wesley 1980.

[Mallgren83] W. Mallgren, *Formal Specifications of Interactive Graphics Programming Lnaguages*, MIT Press 1983.

[Morrissey79] J. Morrissey and L. Wu, "Software Engineering … An Economic Perspective," in *4th International Conference on Software Engineering*, pp. 412-422, September 1979.

[Neighbors80] J.M. Neighbors, *Software Construction Using Components*, Ph.D. Thesis and Tech. Rep. TR-160, University of California, Irvine, ICS Dept., 1980.

[Neighbors84a] J.M. Neighbors, J. Leite, and G. Arango, *Draco 1.3 Manual*, Tech. Rep. RTP003.3, University of California, Irvine, ICS Dept., June 1984.

[Neighbors84b] J.M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, pp. 564-574, September 1984.

[Press86] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 1986.

[PrietoDiaz86] R. Prieto-Diaz and J.M. Neighbors, "Module Interconnection Languages," *The Journal of Systems and Software*, vol. 6, pp. 307-334, November 1986.

[PrietoDiaz87] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, pp. 6-16, January 1987.

[Rich81] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice," in *7th International Joint Conference on Artificial Intelligence*, pp. 1044-1052, August 1981.

[Ross77] D. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, vol 3, pp. 16-34, January 1977.

[Rowe78] L.Rowe and F. Tonge, "Automating the Selection of Implementation Structures," *IEEE Transactions on Software Engineering*, vol. 4, pp. 494-506, November 1978.

[Royce70] W. Royce, "Managing the Development of Large Software Systems", in *Proceedings, IEEE WESCON*, August 1970. reprinted in *9th International Conference on Software Engineering*, pp. 328-338, April 1987.

[Sedgewick84] R. Sedgewick, *Algorithms*, Addison-Wesley, August 1984.

[Simon69] H. Simon, *The Sciences of the Artificial*, MIT Press 1969.

[Smith85] D. Smith, G Kotik, and S. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1278-1295, November 1985.

[Standish76] T.A. Standish, D. Harriman, D. Kibler, and J.M. Neighbors, *The Irvine Program Transformation Catalogue*, Tech. Rep., University of California, Irvine, ICS Dept., 1976.

[Sundfor83a] S. Sundfor, *Draco Domain Analysis for a Real Time Application: The Analysis*, Tech. Rep. RTP 015, University of California, Irvine, ICS Dept., 1983.

[Sundfor83b] S. Sundfor, *Draco Domain Analysis for a Real Time Application: Discussion of the Results*, Tech. Rep. RTP 016, University of California, Irvine, ICS Dept., 1983.

[USDODAda83] United States Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL STD 1815A-1983, U.S. Government Printing Office. Ada is a registered trademark of the U.S. Government (Ada Joint Program Office). February 1983.

[Waters85] R.C. Waters, "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Transactions on Software Engineering*, vol. 11, pp. 1296-1320, November 1985.

[Wile83] D.S. Wile, "Program Developments: Formal Explanations of Implementations," *Communications of the ACM*, vol. 26, pp. 902-911, Nov. 1983.

[Wile86] D.S. Wile, "Local Formalisms: Widening the Spectrum of Wide-spectrum Languages," in *Conference on Program Specification and Transformation*, IFIP Working Group 2.1, April 1986.

[Yourdon79] E. Yourdon and L. Constantine, *Structured Design*, Prentice-Hall, 1979.