The Evolution from Software Components to Domain Analysis

James M. Neighbors
System Analysis, Design, and Assessment

Abstract

More than twenty years ago the idea of producing software systems from reusable software components was proposed. Since that time many changes have taken place in Computer Science and Software Engineering, but software systems are still built as one-of-a-kind craftsman efforts. A method for software construction using components is rationalized using experience from software components, program transformations, system architecture, industrial large systems, automatic programming and program generation. Experience with the method is discussed. The limiting factors of the method that prevent the widespread use of reusable software components are identified.

Introduction

We will focus on schemes for the production of large, quality software systems that can be extended and maintained over a lifespan of many years. The paper extracts requirements for a knowledge-based system constructing system. These requirements are used to rationalize the Draco methodology [Neighbors84b].

There are two primary approaches to producing anything: the craftsman approach and the mass-production approach. The craftsman approach relies on a highly skilled craftsman to build an object from raw materials. The raw materials are fashioned into custom parts and fitted together to form custom assemblies. The mass-production approach relies on prebuilt standard parts and standard assemblies of parts to be combined to form the object. Each of the approaches has its good and bad points.

With the craftsman approach, the custom parts and assemblies are tailored to the specific problem at hand. These custom parts can have a very efficient implementation; probably better than could be built from standard parts. Given the time, a craftsman *always* builds a better object than one constructed from standard parts. By "better" here we mean more responsive to the goals of construction. The craftsman approach has its drawbacks in that craftsmen are expensive to employ and hard to find. Any system built by a craftsman is a custom system and will require custom maintenance. This means that the maintenance must be done by a craftsman who must shape new custom parts to fit with the old custom parts in an object.

The mass-production approach offers cheaper construction costs since the object is built from prebuilt standard parts. An assembly is a structure of standard parts that cooperate to perform a single function. The use of standard parts and assemblies implies the availability some knowledge about the failure modes and limits of the parts. This information is unavailable with custom parts. Use of standard parts also creates a language for discussion of future objects and extensions to objects currently under construction. The mass-production approach has its drawbacks in that the design of useful standard parts and assemblies is very expensive work and requires craftsman experience. Also, once a set of standard parts is created it may not suffice to construct all the objects desired.

Our wish to build software systems from reusable software components represents a shift from craftsman production to mass-production. This shift is forced upon us by the ever increasing size of software systems we build.

Software Components

The idea of constructing software from general, well-specified, and well-tested software components is an appealing one. After all, we software engineers have seen the computer hardware engineers succeed using this technique[1] time after time. McIlroy [McIlroy68] is one of the earliest and most eloquent advocates of software components. He envisioned a complete industry similar to the semiconductor industry with factories solely dedicated to the mass-production of all kinds of software components. These components are cataloged and placed into libraries for ready access.

> "I would like to see components become a dignified branch of software engineering. I would like to see standard catalogues of routines, classified by precision, robustness, time-space performance, size limits, and binding time of parameters. I would like to apply routines in the catalogue to any one of a large class of often quite different machines…What I have just asked for is simply industrialism, with programming terms substituted for some of the more mechanically oriented terms appropriate to mass production." [McIlroy68]

Further, if the idea of software components works well perhaps we could bind them together using other analogies to the hardware world like "busses" and "sockets". If these techniques worked they would move software production out of the craftsman era and into the mass production era. These are brave and intuitive ideas that have not come to pass. Why? It is a goal of this paper to answer that question using experience.

In 1973 I became a project software manager in a company that specialized in selling custom real-time, high-speed data acquisition and control systems. I had read McIlroy's vision of software components and became convinced that constructing systems using software components was the way to go. I asked the programmers on my project to extract components from the systems they had built. Programmers throughout the company became interested and submitted components. A programmer had to submit a component to get a copy of the catalog and the object module library. It was not a restriction. It was more a matter of pride.

Please address correspondence to James M. Neighbors, Systems Analysis, Design, and Assessment (SADA), Box 5017, Irvine, CA 92716, USA

1. Later we will argue that hardware and software engineering only *appear* similar. However this early view did motivate work on software components.

The company made data acquisition hardware so the first wave of components were drivers for the hardware. All the work was in assembly language so the components were assembled, cataloged, and placed in an object module library. The second wave of components were assembly language routines that came from the computer manufacturer to perform useful functions like emulate the floating point hardware, string handling, formatted printing, math functions, etc. At this point we could snap together a simple and not very fast system. The third wave of components came from a completely unexpected source – the senior systems analysts. Systems analysts specified what the systems did. Programmers simply made the hardware do that. Project software managers and not programmers talked to systems analysts. There were very few of them (3 per 25 programmers). They had been in this business for many years and only got involved with an actual system to fix a mess. They submitted the most wondrous components! They were "tricks" that really made the systems fast. The following are some of the components submitted by the systems analysts:

1. Methods for using the timers to interrupt before the data interrupt to avoid the interrupt context switch time.
2. Optimal interleaving of sensor data requests and reads to avoid data settling time.
3. Arcane algorithms for converting synchro bit data to angles without sine and cosine tables.

As programmers who had experienced these problems and did not invent these answers we quickly made these techniques a part of our repertory and systems.

The software component library was a success over the traditional craftsman approach taken by the company. We produced small systems (5000 assembly lines) that interfaced to other systems mostly as equipment (switches and sensors). Ultimately, we reduced the time to build a new custom system in this constrained domain to 20% of the craftsman development time. However problems began to appear with the software component library. As the "inventor" of this concept at the company I became the agent for finding and modifying components from the library. In this role as a librarian certain problems with the library became apparent. In some cases a programmer was looking for a program part that could just be "plugged in" without change. In other cases the programmer was looking for a program part that could be changed before use. As an example, a senior analyst had submitted a component that calibrated and accessed a nonlinear temperature sensor with 0.01 degree accuracy through a very complex interpolation. A programmer with a new application only needed 0.5 degree accuracy at higher speed. Neither of us knew how to change this complex component. This is an important consideration in the design of a library of reusable program parts. What a part does only allows its reuse without change. What a part does, how it does it, and how changes may be made allows the reuse of a component with change.

One straightforward way of organizing a collection of software parts is to put each part into a library of source code indexed by the "what" description of each part. 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 a proper 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 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.

A software component library offering components that can be changed before use must store "how" information in addition to "what" information for each part. This "how" information describes how the part performs its function and how changes are made. Organizing a library allowing change will encounter the following additional problems:

1. *structural specification problem*: What is a proper 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.

Within the context of the existing tools at the company (text editors, linkers, and object module libraries) we pushed the software component library concept to its limit. I entered graduate school hoping that Computer Science could solve my library problems.

In graduate school I learned of projects [Corwin72, Campos78] that not only had tried software component libraries but had extended the hardware analogy to include "sockets" and "busses" between the components. This let them characterize and type the data flowing between the modules. This work suffered from the same general library problems I had met in building small component libraries. However, a more ominous problem occurred to me as I read how their assembly mechanism assembled and checked the component interconnections. Inherent in all the software component work is that (in McIlroy's vision) the component business will "scale up" to cover all aspects of software production on all sizes of systems. The *library problem* limits the straightforward idea of software component libraries from scaling up:

> "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 while increasing the classification and searching problems. If a library contains a small number of large parts, then it lessens the classification and searching problems while increasing the structural specification and flexibility problems." [Neighbors80]

A successful software component library would contain millions of tiny components. The data passed along "busses" and "sockets" between components changes at component use time rather than being fixed at component creation time. How could such a library be organized? The "library problem" stopped the mass production of software components.

Working on the data acquisition and control systems gave me respect for people like the senior systems analysts who knew how your system worked before you explained it to them. Later I would come to call them "Domain Analysts" because they understand how the entire class of systems that addresses a specific problem domain should work. As with the acquisition and control systems domain, the domains are never found in books – perhaps they should be but it is hard to classify material about some problem domains. Most problem domains are so case specific that no overriding structure is yet discovered on which to base a book. They are in a state similar to compiler theory during the 1950s. The basic structure has not been identified. Domain Analysts make the systems of the world work in the absence of widely accepted structural models.

Lessons from Software Component Library Research:

1. Libraries have an immediate impact and are a success but they do not solve the larger problem. The "library problem" prohibits simple software component library schemes from scaling up to larger problems.
2. The reuse of program parts without change is extremely successful. The implementation of compilers by linkage to run-time support routines is an obvious example of this technique.
3. The reuse of program parts changed by programmers is a major activity of detailed design and coding. Encyclopedic works such as [Knuth68, Sedgewick84, Press86] are successes because they serve as guides supplying information above the level of programming language code. This 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.
4. Synthesizing components for sorts, list insertions, and most operations on numbers are not the problem. They may be used as research examples, but if that is the extent of the work, then the reader should be wary of whether the work will "scale up" or not.
5. Hardware analogies such as "busses" and "sockets" constrain software. Software components pass more complex structures than hardware components. For software components the structure of information passed through interface points changes *at component use time*. For hardware the structure is usually a fixed standard declared *at component creation time*.
6. "Domain Analysts" are a wealth of formal and informal experience about how systems in the domain *actually* work. Any successful technique for building systems in a problem domain must have a method for gathering and using this valuable information.

Program Transformations

I became interested in program transformations as a way to introduce flexibility into source code software components. I believed that very general components, such as the high accuracy temperature sensor component discussed earlier, might be transformed into different lower accuracy versions dynamically without having to store those versions explicitly. If this could be done, it would aid the "library problem" by reducing the number of components in the library.

Source-to-source program transformations treat a program as an algebraic object with rewrite rules. Each transformation has a left-hand pattern (LHS), a right-hand pattern (RHS), and enabling conditions (EC) on the pattern variables [Standish76]. A simple transformation would be:

```
LHS: X*(IF P THEN A ELSE B) <=>
RHS: (IF P THEN X*A ELSE X*B)
EC: X and P are execution order independent
```

The flavor of source-to-source transformations can be experienced by transforming a simple matrix multiply [Kibler77].

```
FOR I:=1 STEP 1 UNTIL N DO
  FOR J:=1 STEP 1 UNTIL N DO
    BEGIN
      C[I,J]:=0;
      FOR K:=1 STEP 1 UNTIL N DO
        C[I,J]:=C[I,J]+A[I,K]*B[K,J];
    END;
```

Now assert that matrix `A` is the identity matrix using an equation for the values of `A` as:

```
A[row,col] -> (IF row=col THEN 1 ELSE 0)
```

The original matrix multiply is rewritten as:

```
FOR I:=1 STEP 1 UNTIL N DO
  FOR J:=1 STEP 1 UNTIL N DO
    BEGIN
      C[I,J]:=0;
      FOR K:=1 STEP 1 UNTIL N DO
        C[I,J]:=C[I,J]+(IF I=K THEN 1 ELSE 0)*B[K,J];
    END;
```

General program transformation rules that apply to assignments, loops, and arithmetic can specialize this program. There are about 30 low-level transformations applied. The major steps in the transformation of the inner loop are shown below.

```
C[I,J]:=0;
FOR K:=1 STEP 1 UNTIL N DO
  C[I,J]:=C[I,J]+(IF I=K THEN 1 ELSE 0)*B[K,J];

C[I,J]:=0;
FOR K:=1 STEP 1 UNTIL N DO
  (IF I=K THEN C[I,J]:=C[I,J]+1*B[K,J]
          ELSE C[I,J]:=C[I,J]+0*B[K,J]);

C[I,J]:=0;
FOR K:=1 STEP 1 UNTIL N DO
  IF I=K THEN C[I,J]:=C[I,J]+B[K,J];

C[I,J]:=0;
IF (I>=1) AND (I<=N) THEN
  C[I,J]:=C[I,J]+B[I,J];
```


3

```
C[I,J]:=0;
C[I,J]:=C[I,J]+B[I,J];

C[I,J]:=B[I,J];
```

The final version of the matrix multiply where matrix A is the identity matrix becomes a matrix copy as expected.

```
FOR I:=1 STEP 1 UNTIL N DO
  FOR J:=1 STEP 1 UNTIL N DO
    C[I,J]:=B[I,J];
```

To perform this simple transformation at the low level of abstraction of an algorithmic programming language the transformation system has to use many transformations from a large space of possible transformations. This is a big search problem and hard work. Planning sequences of these simple low-level transformations can require a lot of AI planning [Fickas85].

Now consider this same matrix multiply example in a language like APL that embraces the concept of matrices and matrix operations. This new language is at a higher level of abstraction than the usual algorithmic programming language. Given that 1_{mat} is the identity matrix in this new language the identity matrix multiply becomes the simple transformation.

LHS: $C_{mat}:=1_{mat}*B_{mat}$ <=> RHS: $C_{mat}:=B_{mat}$
EC: structures of C and B are equal

Notice that this is similar to the transformation for multiplying integers or reals by 1 in algorithmic languages. From this simple example we can see that a lot of hard work can be avoided by creating levels of abstraction above algorithmic languages that directly define the concepts of interest.

The example above does not advocate doing away with the low-level algorithmic transformations. They serve a useful purpose in the optimization of general programs under arbitrary conditions. Consider the complex optimization performed by the transformations under the assertions that A is an upper-triangular matrix and B is a lower-triangular matrix.

```
A[row,col] -> (IF row<col THEN A[row,col] ELSE 0)
B[row,col] -> (IF row>col THEN B[row,col] ELSE 0)
```

In some cases if a transformation is not performed at a high enough level of abstraction then the effect of the transformation may never be achieved. Consider the case of an algorithmic language and an exponentiation operator (**). If the phrase X**2 were encountered in a program we could employ the simple source-to-source transformation

LHS: X**2 <=> RHS: X*X
EC: X is side-effect free

to convert it to multiplication; or we could macro expand a general implementation of the exponentiation operator and then try to simplify. The "binary shift method" is a general expansion of the exponentiation operator when the power is a positive integer. The macro expansion of X**2 using the binary shift method is shown in figure 1.

```
BEGIN
  POWER:=2; NUMBER:=X; ANSWER:=1;
  WHILE POWER>0 DO
    BEGIN
      IF ODD(POWER) THEN ANSWER:=ANSWER*NUMBER;
      POWER:=POWER SHIFT_RIGHT 1;
      NUMBER:=NUMBER*NUMBER;
    END;
  RETURN ANSWER;
END;
```

Figure 1: Implementation of X**2 using Binary Shift Method

The "Taylor expansion method" is a general expansion of the exponentiation operator where the number raised to a power must be positive. The macro expansion of X**2 using the Taylor expansion method is shown in figure 2.

```
BEGIN
  SUM:=1; TOP:=2*LN(X); TERM:=1;
  FOR I:=1 TO 20 DO
    BEGIN
      TERM:=(TOP/I)*TERM;
      SUM:=SUM+TERM;
    END;
  RETURN SUM;
END;
```

Figure 2: Implementation of X**2 using Taylor Expansion

The "binary shift method" expansion may be reduced to a simple multiply by chaining together many low-level algorithmic language source-to-source transformations similar to the process of transforming matrix multiply by the identity matrix. The "Taylor expansion method" expansion cannot be reduced to a simple multiply by general low-level transformations because it is an *approximation* of exponentiation. It suffices as an implementation because of its context (in this case that 20 terms of accuracy is acceptable) but it is not equivalent. Other investigators in this area ran into the same problem.

> "We are able to make full use of the algebraic laws appropriate to this higher level. For example, once calls to set operations have been replaced by their list processing bodies many possibilities for rearrangement and optimization will have been lost." [Darlington73]

These examples show that very simple mechanisms (source-to-source transformations) applied at a higher level of abstraction can exceed in power very complex mechanisms (AI planning and dataflow analysis) applied at lower levels of abstraction. Some optimizations are no longer possible as we go to lower levels of abstraction. Level of abstraction knowledge about the problem domain is more powerful than general mechanisms.

The reader might well ask "Who would write programs containing such statements as X**2?" Systems that combine very general software components create such statements all the time. They reflect generality that is not being used in a particular case. The role of source-to-source transformations is to smooth out this generality using a simple mechanism on concepts at a high level of abstraction. Any work that seriously uses layers of knowledge abstraction will employ simple source-to-source transformations for optimization.

I investigated a simple scheme of Markov processes that provides a procedural capability with proof of termination for source to source transformations [Neighbors80]. This scheme is useful for transformations that must propagate or use global information.

Lessons from Program Transformation Research:

1. There are few, if any, equivalence preserving transformations. This is not a problem as exemplified by optimizing compilers. Correctness preserving transformations in a given context are the issue.
2. Using concepts at the "right" level of abstraction is an extremely powerful optimization technique. This represents a tradeoff between planning and knowledge.
3. The rules of exchange in a domain must be absolute with respect to the semantics of the domain. This means the rules apply independent of any implementations chosen for the domain. The granularity of the semantics of a domain only applies to statements in the domain – not implementations.

System Architecture

Software Engineering system architecture theories gave me the tools to cope with complexity. If software components were ever to be a success, clearly something beyond including millions of components into a flat catalog must be the goal. The early Software Engineering discussions on levels of abstraction provided very strong ideas.

> "We understand complex things by systematically breaking them into successively simpler parts and understanding how these parts fit together locally. Thus, we have different levels of understanding, and each of these levels corresponds to an *abstraction* of the detail of the level it is composed from. For example, at one level of abstraction, we deal with an integer without considering whether it is represented in binary notation or two's complement, etc., while at deeper levels this representation may be important. At more abstract levels the precise value of the integer is not important except as it relates to other data." [Knuth74]

The problems with building large software systems in the late 1960s prompted the study of how systems are produced. The discussion at the 1968 and 1969 NATO conferences focuses on process and abstraction. Suddenly there was a lot of thought about how large systems are partitioned into parts and how these parts are interfaced. Later research programming languages such as Clu and Alphard incorporated the abstraction idea and provided the ability to create component interfaces stronger than "sockets" and "busses". The result of partitioning a system into parts became known as the architecture of a system. Tools that produce code using software components create system architectures either implicitly or explicitly.

System architecture is how a system is structured to perform its function. For a specific system there is only one system function but there are many architectures that can provide that function. The architecture is separate from function. The basic tenet of good design is that a system architecture should follow the decomposition of the system function. This technique breaks down when we stop modeling the objects and operations of the problem domain and start using known Computer Science abstractions to model the problem. The closeness of the top levels of architecture and function sometimes leads to their confusion.

In Software Engineering there are two basic approaches to developing system architecture: stepwise refinement [Wirth71, Dijkstra69] and layers of virtual machines [Dijkstra68]. Strict stepwise refinement stresses the decomposition of a system:

> "A guideline in the process of stepwise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible." [Wirth71]

Strict stepwise refinement results in architectures that are tree-like as functions are subdivided into separate subfunctions. The module reference structure of a system produced using stepwise refinement might appear as shown in figure 3.

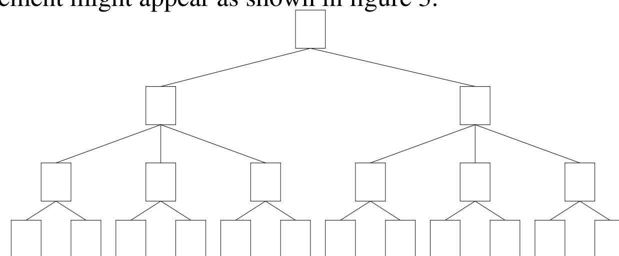

Figure 3: Stepwise Refinement Architecture

Inherent in the stepwise refinement model is the assumption of flexibility at the bottom of the architecture. The primary constraining factors come from higher levels of abstraction.

Creating architectures from layers of virtual machines was described by Dijkstra.

> "Phrasing the structure of our total task [build a multiprogramming operating system] as the design of an ordered sequence of machines provided us with a useful framework in marking the successive stages of design and production of the system. But a framework is not very useful unless one has at least a guiding principle as to how to fill it in. Given a hardware machine A[0] and the broad characteristics of the final machine A[n] (the value of 'n' as yet being decided) the decisions we had to take fell into two different classes:
>
> 1. we had to dissect the total task of the system into a number of subtasks
> 2. we had to decide how the software taking care of those various subtasks should be layered. It is only then that the intermediate machines (and the ordinal number 'n' of the final machine) are defined.

> Roughly speaking the decisions of the first class (the dissection) have been taken on account of an analysis of the total task of transforming A[0] into A[n], while the decisions of the second class (the ordering) have been much more hardware bound."[Dijkstra68]

Following the above prescription results in architectures that have some functional decomposition but are primarily organized as layers of implementing function. The module reference structure of a system produced using layers of virtual machines might appear as shown in figure 4.

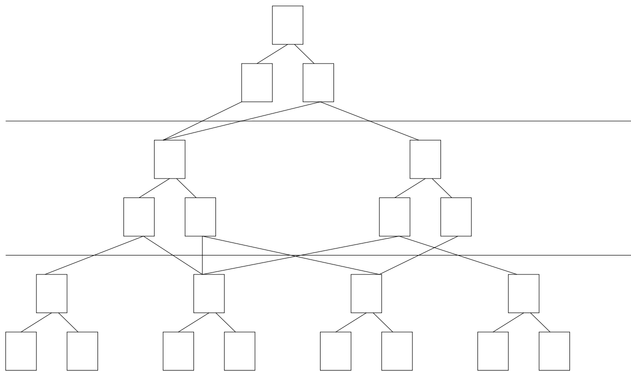

Figure 4: Levels of Abstraction Architecture

Inherent in the layers of virtual machines model is the assumption of flexibility at the top of the architecture. The primary constraining factors come from lower levels of abstraction.

Stepwise refinement focuses on creating architectures as the functional decomposition of the system function. It partitions the call graph of system modules vertically. The layers of virtual machines approach focuses on creating architectures that provide strongly defined layers of abstraction. It partitions the call graph of system modules horizontally. Though these two approaches to architecture are opposed (one suggesting vertical partitioning and one suggesting horizontal partitioning) there is agreement. Both approaches stress the need for encapsulation and simply suggest two methods for determining the next unit of encapsulation.

Real programs of course use both methods and result in module reference structures that might appear as shown in figure 5.

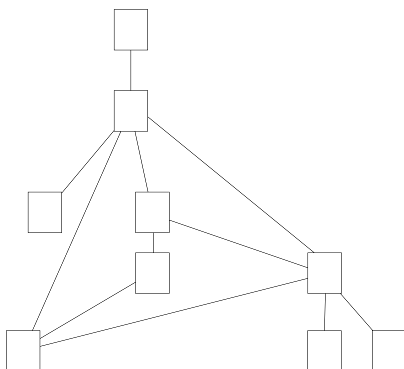

Figure 5: Real Program Architecture

Later work [Parnas72] introduced the principle of maximal "information hiding" as a criteria for determining which approach to use in the successive steps of developing an architecture.

The reader may ask "What does all of this discussion about architecture have to do with generating software using software components?" The ability of a program generation system to produce variations in architecture indicates an ability to create and use abstractions. Ultimately, all generated programs must use abstractions imposed on them from the outside world (e.g., file systems, graphics systems, database systems). These abstractions are not only useful for structuring the system but they can also be used to explain the developed system to people. I showed that changing the architecture of a system can completely change the time and space characteristics of the system function [Neighbors80]. This is not a big secret. Programmers have been instantiating procedure bodies inline for years to gain execution speed. For these reasons we should be suspicious of program generation systems that only address system function and don't address system architecture. What do they provide as an architecture?

Lessons from System Architecture Research:

1. System architecture exists and it is separate from function.
2. System architecture has a big impact on the performance and maintainability of a system.
3. Encapsulation mechanisms such as packages and objects are used to create system architecture.

Large Systems

In Software Engineering there was a lot of discussion of how abstraction and typing mechanisms would enable us to build large (million source code line) systems. There was very little examination of large systems to determine how the developers of these systems had survived all these years without the new abstraction mechanisms. After all, large systems did exist. How did they get them to work?

My curiosity led me to specialize in Software Engineering techniques applied to industrial large systems. The industrial organizations tolerated me because I could translate proven Software Engineering findings into the organization. Organizational infrastructure issues such as coding standards, lifecycle models, management tools, document control, version control, and configuration management are vital for industry. At the same time I made it an issue to talk to everyone involved with a large system and to scan the actual source code of the systems.

It is impossible to examine the source code of a large system by hand. A million line system may have as many as 8000 modules! Examining 40 modules per day (5000 lines per day) it would take a complete year to examine each module. Large systems are usually old systems. It takes a long time for a system to grow to a million lines. Typically a million line system is between 15 to 25 years old. They are written in the most widely used languages at the time, FORTRAN and COBOL. For my examination of the structure of these systems I use code auditing and source code scanners based upon

metacompilers [Schorre64]. The source code scanners scan the entire source code, write reports, and propose areas for code auditing.

The developers of large systems get them to work by very carefully controlling interconnections between components in the system and the usage of global resources. Some global resources are surprising. As an example if the system contains 8000 modules, the space of *module names* is controlled and has a pattern. A module named `MSRC10` might be a module that deals with receiving (`RC`) messages (`MS`) of type `10`. The obvious solution to this problem is to allow larger names. If you do this the programmers concatenate architectural structure names onto the routine names. This is fine while the architecture does not change. When the architecture does change no one wants to go back and change all the now misleading names. Increasing the size of names does not improve the name space problem.

Consider the interconnection in figures 6 and 7 drawn from a group of three systems (about 4 million source lines in 11,000 modules) in FORTRAN and Pascal. The bars represent the range and median values on the logarithmic percentage scale. The architectures of these large systems are very vertical. Figure 6 indicates that more than half of the modules in a large system only exist in the context of the one module that calls it. This provides a strong clue to the problem of understanding large systems. Establishing the context of each module and conceptually collapsing modules that exist only in the context of one module will enable us to conceptually remove more than half of the system modules!

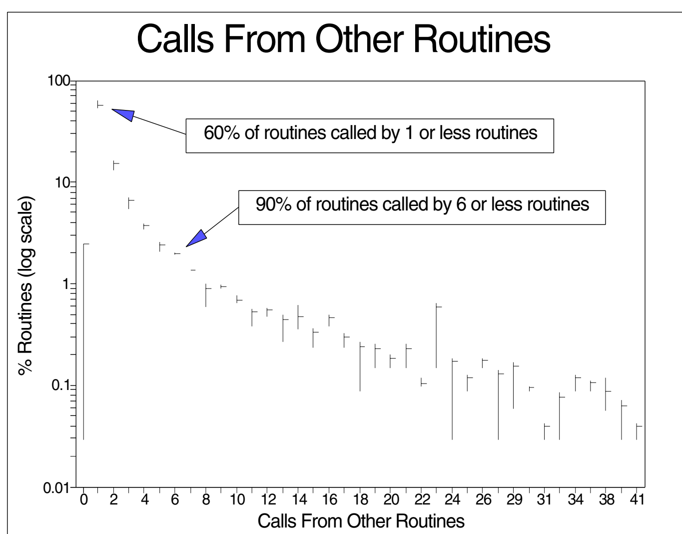

Figure 6: Number of Calls From Other Routines

The number of calls to other routines continues the careful partitioning of a large system. Figure 7 indicates that about half of the routines only call three or fewer routines. These systems contain thousands of modules. Clearly the tradeoff between using a part of the "name space" to create a new routine and encapsulating information in routines is taken very seriously.

Large systems are rare. They are evolutionary survivors. For each 20 year old large system there were many competing systems that could not grow to this size. These systems are expensive to maintain and evolve. I have found that one programmer is required for each 10,000 to 30,000 lines of source code. At a burdened man-year cost of $90,000 to $150,000, a million line system costs between $3 million and $15 million per year to maintain. These systems must earn their keep every year or die. They are kept alive by careful partitioning of the system functions and maintenance of the partitions.

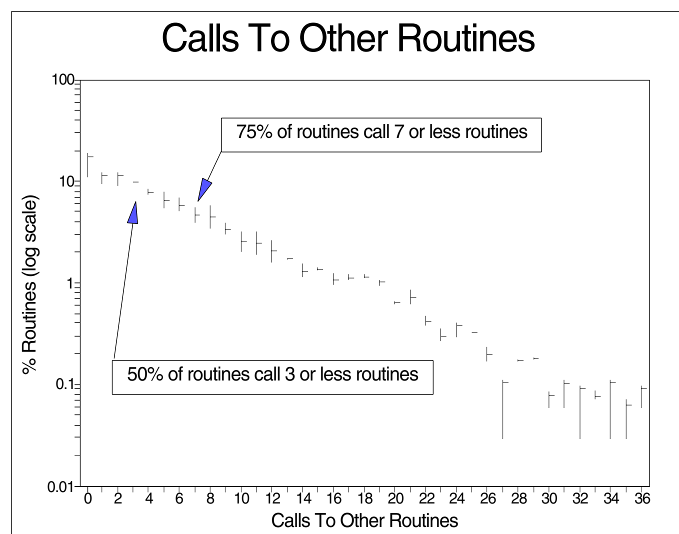

Figure 7: Number of Calls To Other Routines

Sometimes the development of a large system gets out of control. This manifests itself in a variety of ways:

- Inability to add new features.
- Inability to correct errors without introducing errors (critical mass).
- Inability to get a consistent build of the entire system.
- General agreement that the system is a "spaghetti code mess."

It is surprisingly easy to bring such a system back under control. First, no system gets to be this large while truly being a "spaghetti code mess." The developers are really saying that they do not understand how the system fits together anymore. The steps to bring the system under control are:

1. Make sure there is about one programmer per 20,000 lines of source code.
2. Identify tightly coupled modules in the source code. The coupling should include definition, control, data, and message coupling.
3. Form these tightly coupled modules into subsystems and identify the subsystem's interface and responsibilities to the rest of the system.
4. Assign 10,000 to 30,000 source code lines worth of subsystems to each programmer.

I have found that even though there is a constant lines of code per programmer ratio the programmers are not assigned specific

7

sections of code approximating this size. Instead the systems are loosely divided into five to ten large chunks. A programmer is expected to work in two or more chunks. This provides the management with the security that the loss of a single programmer does not leave any codes uncovered. This costs the management in that the programmers must fall back into a kind of large system "maintenance programming." This form of programming carefully brackets changes by `IF-THEN` clauses to make sure that the change doesn't introduce errors. The programmers do not understand the context in which each code is called. After many years of this kind of change, the systems are very hard to understand.

Assigning a subsystem with an explicit interface to an individual programmer changes the programmers' outlook on the code. From the interface definitions and interconnection analysis the programmer knows the context in which each routine is called. The code actually begins to shrink as special cases built into the code over the years are identified as no longer in use (dead code) and removed. A "pride in ownership" sets in as the programmer realizes that if he carefully cleans up his subsystems it will make his job much easier. He, personally, will benefit from this work. This is a powerful new incentive.

Massive change occurs in large systems. A large percentage of the modules are changed every year by the supporting programmers. It is the trick of large system management to harness the massive change to improve the system. I have found assigning subsystems to individual programmers to be successful in achieving an improvement in system structure and reliability.

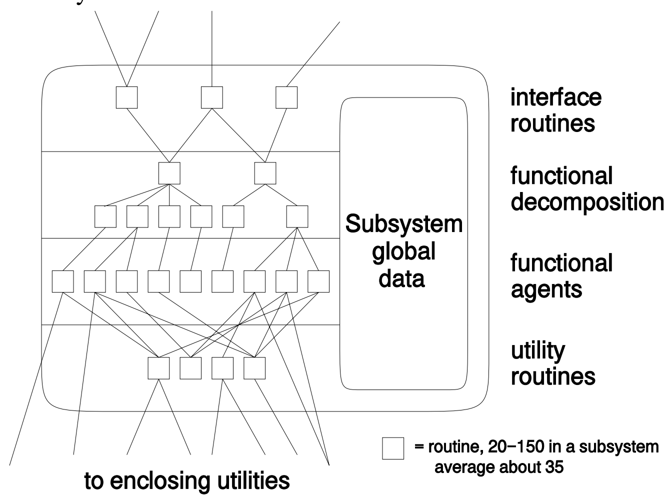

Figure 8: Model of Subsystem Structure

What do these subsystems look like? Once again this is not very surprising but they look like small systems embedded in a large system. The subsystem is partitioned by both decomposition for its interface function and layers of abstraction for utility support. Figure 8 shows the structure of a typical subsystem. The structure of a large system is very vertical except for global routines that manage the database. It seems that all large systems are database systems because the data they manage is too large to fit into main memory. The database routines are always the most called routines in a large system.

Once we have partitioned our system into subsystems and assigned them to individual programmers is our large system under control? Well, no it is not yet. Extremely large systems are over 10 million source code lines and over 30 years old. Systems of this size contain hundreds of subsystems. Each subsystem makes its interface public but we do not want just anyone calling a subsystem. New module encapsulation schemes such as Ada packages and Object-Oriented Programming (OOP) do not suffice to build large systems. Both are valuable encapsulation mechanisms and should be used to define global system resources. However, they only provide information about the resources. Once an object or package is declared any other object or package may use it. All system resources become global. Both of these encapsulation mechanisms have difficulty with global issues. The Ada package users have run into the problem of having to form multiple packages into higher level groups to provide an abstraction.

> "Packages are a *necessary* mechanism in the decomposition of Ada systems…However, packages are not a *sufficient* mechanism for decomposition or reusability. The reason for this is that there are some abstractions that are simply too intellectually large to be conveniently captured in a single package." [Booch87, pg. 556]

Similarly, OOP does not discuss how descendant objects are constrained. As an example, all graphic objects must have a method of rendering, but it cannot be inherited since there is no global method of rendering for all graphics objects. Any inheritance here is an error. We must define a graphics object that *requires* all descendant graphics objects to define their own rendering. This is a global resource required of all graphics descendant objects.

OOP and package-like encapsulations do not provide information about the control and flow of resources in the total system. Module Interconnection Languages (MILs) were designed to provide this important architectural function for systems with many subsystems [DeRemer76, Cooprider79, Tichy79, Prieto-Diaz86]. MILs form the resources presented by the subsystems into an architecture for the overall system. MILs are based on the difference between programming-in-the-large (PL) and programming-in-the-small (PS).

> "Structuring a large collection of modules to form a system [PL] is an essentially different intellectual activity from that of constructing the individual modules [PS]" [DeRemer76].

Architects of a large system are primarily concerned with the process of composing system modules rather than with the process of programming each module.

PS is concerned with building modules using conventional programming languages. It focuses on how a particular part (module) of a system performs its function. PL is concerned with building systems. It focuses on how the system modules cooperate (through calls and data sharing) and what functions each module provides. The MIL specification of a system is a formal written description of the system architectural design. A version of the system must conform to this description before it can be constructed. *A maintenance programmer cannot knowingly or unknowingly violate the system design without explicitly changing the system design.*

The MIL specification of a complete system must include three items:

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

In a MIL description, resources are any entity in a PS programming language (e.g., variables, constants, procedures, type definitions, etc.) that can be made available for reference by other modules. Many modern PS languages bundle this PL resource information with separately compilable units (packages, modules). This may prohibit PL interconnection checking without PS information.

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

```
module ABC
    provides a,b,c
    requires x,y
    consist-of function XA, module YBC
        function XA
            must-provide a
            requires x
            has-access-to  module Z
            real x, integer a
        end XA
        module YBC
            must-provide b,c
            requires a,y
            real y, integer a,b,c
        end YBC
end ABC
```

The subsystems derived from large systems should be cast in this form to guarantee the validity of the system architecture during maintenance.

People have found the concept of subsystems to be important in large system development. As with modules and other encapsulation mechanisms, this concept should be an important aspect of architecture for a tool that builds large systems out of software components. In fact, as with hand-built systems, the tool might find predefined subsystems a useful method of reasoning about the system under construction.

Lessons from Large System Research:

1. To learn about large systems you must actually look into large systems. Primarily large systems of a million source lines or more are found only in industry. Experience with 10,000 source lines and below does not translate well into the large system arena.
2. System architecture is very important in large systems. Programming-in-the-small structures (OOP, packages, modules) are different from programming-in-the-large structures (MILs). MILs are required to control the use of encapsulated abstractions.
3. Subsystem architecture erodes as the system is maintained. Finding existing components in a large existing system must deal with this issue. Bringing large system development under control entails re-establishing the architecture and assigning responsibilities with respect to that architecture. Assigning subsystems with established and defensible interfaces to individual programmers promotes pride in ownership. The method harnesses the force of change on the system.
4. A mechanism that constructs systems from reusable components must address the issue of architecture. Architecture can drastically change the execution and space requirements of systems.

Automatic Programming and Program Generation

I became interested in automatic programming and program generation because these areas took the idea of levels of abstraction right up to the user's problem domain.

> "A model of the problem domain must be built and it must characterize the relevant relationships between entities in the problem and the actions in that domain." [Balzer73]

Software Engineering explained the idea of abstractions and decomposition much better than Artificial Intelligence described the partitioning of knowledge nets. Software Engineering then focused on the lower-level bottom-up abstractions such as abstract datatypes. Artificial Intelligence focused on high-level top-down abstractions such as relationships in the real world. Software Engineering generated layers of abstraction (abstract datatypes) and automatic programming generated decompositions (instantiated knowledge nets).

I was very much impressed by the power of program generators[2] that actually produced successful application programs from high-level domain-specific descriptions of the problem. Program generators are very narrow in their scope of application – usually business data processing in COBOL. They rely on a broad rigid model of the problem domain with a very simple mechanism to assemble the resulting code. Often simple conditional macro expansion from an assembler is used! The knowledge about the problem domain is held as text strings in macro bodies. This is similar to the "sysgen" procedures of early operating systems.

The power of program generators was not lost on the automatic programming community.

> "The people who work in this area [automatic programming] fully realize that for practical solutions, their ideas will have to be combined with those of the first type [program generation], incorporating specific knowledge of the domain being treated." [Feldman72]

2. Program generators were later to be known as 4th Generation Languages (4GLs).

9

From a formal theory standpoint automatic programming had been shown to be a solvable problem [Green69] using theorem proving. However, the computational complexity of theorem proving makes the technique impractical. This early experience with formal theory complexity problems may have pushed automatic programming towards knowledge-based approaches.

To me this was a quandary. On one hand program generation is the most powerful technique for generating software. It is a knowledge-based technique that uses a rigid model of the problem domain. Program generators use very simple mechanisms to construct the software and actually construct real software systems. On the other hand automatic programming techniques use very flexible knowledge representations and very complex planning mechanisms. Their mechanisms extract details from the knowledge net to produce small toy programs.

Knowledge specific to the problem domain is very powerful. It is better to have specific knowledge about the problem domain and a weaker mechanism than a more powerful mechanism and general knowledge. We saw the same effect in program transformations. Most current automatic programming research still prefers to focus on stronger mechanisms and general knowledge schemes.

Lessons from Automatic Programming Research:

1. Problem domain specific specification languages are successful and very powerful. Program generators and 4GLs prove this and are widely used.
2. Domain-specific knowledge-based systems with weak mechanisms have been more effective than strong mechanisms (theorem proving, planning) with weak (general) knowledge bases.
3. The power of the refining (component assembly) mechanism must be carefully balanced against the ability to plan refinement using the mechanism.

Methodology

Our goal is to construct software using components. We need to form the lessons from the techniques we have examined into requirements for a tool that will do this. The primary requirements and their rationale are listed below. Each requirement lists the concept and section in this paper that motivates the requirement.

Requirements(motivation):

1. The tool must accept a description of the objects and operations of a problem domain (domain analysts from software components; decomposition from system architecture; knowledge-based power from automatic programming).
2. The description of a problem domain must be described in terms of problem domains already known to the tool (decomposition and layers of abstraction from system architecture).
3. Optimizations are characterized and performed at each layer of abstraction (optimizing information loss in refinement from program transformations).
4. The burden of search for either implementations (refinements) or optimizations must not be placed on the end user. The tool must suggest implementations and optimizations in the context of the problem (library problem from software component libraries; picking the right transformation from program transformations).
5. The implementation and optimization mechanisms must be computationally tractable so higher-level plans may understand and use their power (simple mechanism capability from program generation; planning from automatic programming).
6. The implementation (refinement) mechanism must provide a wide variation in system architectures to produce a wide variation in time-space tradeoffs in the resulting systems (performance and architecture from system architecture; importance of system structure from large systems).
7. To build large systems and partition system construction, the tool must characterize and generate code that interfaces to existing pre-refined systems (tedium of applying transformations over and over from transformations; the existence of subsystems from large systems)

To address these requirements we have proposed a different methodology of building systems. This has been called the “Draco Methodology” after the first system that we built that used this approach [Neighbors84a, Freeman87]. The organizational dataflow of the method is shown in figure 9.

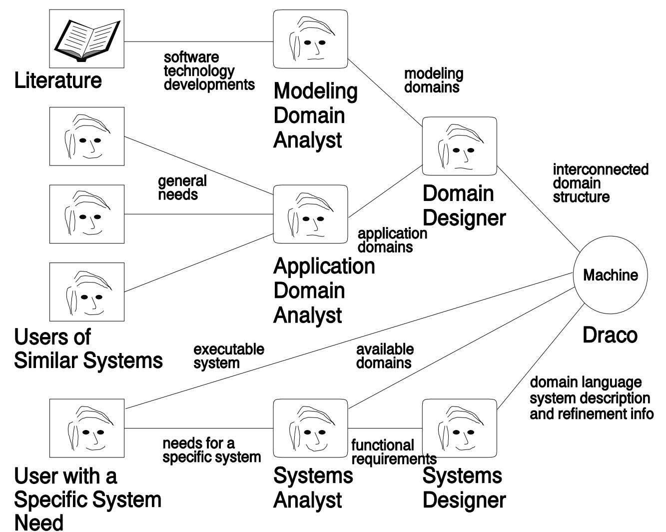

Figure 9: Draco Methodology Dataflow

Using this method we capture a model of a class of systems from the “Application Domain Analysts” who know how application systems of the type are constructed. This is coupled with modeling techniques drawn from Computer Science as understood by a “Modeling Domain Analyst”. This top-down and bottom-up information is combined by a “Domain Designer” to specify a problem domain to the tool. An individual system is specified by a “Systems Analyst” by stating the needs of the specific system in a problem domain known to the tool. If this cannot be done, then the method fails.

In this section we will briefly describe the results of domain analysis and domain design that must be given to the Draco tool to specify a complete domain. These are given in more detail in [Neighbors89, Neighbors84a, Neighbors84b, Neighbors80]. There are six parts to a domain description:

Parser
The parser description defines the interface between the domain and the mechanism. There are four parts to the parser: external syntax (BNF), semantic checks, internal form tree, and database schema for the domain.

Printer
The printer description defines how to communicate domain-specific information to the user. This is necessary for the mechanism to be able to interact with users in the language of the domain and discuss the developing system.

Optimizations
The optimizations represent the rules of exchange between the objects and operations of the domain. Optimizations only work within the domain. There are three parts to the optimization specifications: source-to-source optimizing rules, source-to-source optimizing procedures, and optimization application scripts. The optimization application scripts are plans of optimizations defined for the domain. 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 can be guaranteed since optimizations do not cross domain boundaries.

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 has one or more refinements that represent the different implementations for the object or operation. Each refinement is a restatement of the semantics of the object or operation in one or more domain languages known to Draco. Thus, component refinements cross domain boundaries.

Generators
Generators are domain-specific procedures that are used in circumstances where the knowledge to do domain-specific code generation is algorithmic in nature. This is similar to program generators. The generated programs are kept in the internal form described by the parser description. The construction of optimal hash functions is an example of a generator.

Analyzers
Analyzers are domain-specific procedures that gather information about an input instance of domain notation. The information is kept in a database under the schema defined in the parser description. Dataflow analyzers, execution monitors, theorem provers, and design quality measures are examples of analyzers. Their results are used to check preconditions on refinements and optimizations. They also provide guidance information for user interaction during development.

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

1. *Print* the internal form into the external syntax of the domain.
2. *Optimize* the internal form into a statement in the same domain language.
3. Input the internal form to a program *generator* that restates the problem in the same domain.
4. *Analyze* the internal form for possible leads for optimization, generation, or refinement.
5. Implement the internal form using *software components* each of which has multiple *refinements*. Refinements make implementation decisions by restating the problem in other domain languages.

For *every* problem domain there is a different textual language. People deal with jargon and notation all the time. It is the experience of automatic programming that people have no problem learning a new notation if it helps to solve their problem.

> "There are many large groups of computer users who would be willing to use an artificial language if it met their needs." [Feldman72]

Domain-specific artificial languages like SQL and BNF are easily understood once their notations are defined. The Draco methodology exploits this uniquely human language capability.

> "It is a frequent misunderstanding that there is a separate category of languages called *application-oriented*. In reality, *all* languages are application-oriented, but some are for larger or smaller application areas than others." [Sammet76]

Using specialized languages is an alternative to using program libraries. The languages serve as a general description that limits how the software components of the domain may be combined. Consider FORTRAN not as a programming language but as a surface description scheme for combining the software parts that 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"? A library would not have been as successful because the burden of using the library and knowing the interconnection limitations is placed upon every potential user of the library. Having a domain-specific language that ties the library together removes this burden at the expense of learning the language.

It is easiest to think of the Draco refinement mechanism as the simple macro expansion of a program generator and the optimization mechanism as simple source-to-source transformations applied to domain-specific languages. To provide variable system architectures and consistent implementation choices, some complexity must be added to these mechanisms [Neighbors80]; but these simple models of the mechanism serve to judge the power of the technique.

Experience

Most of the practical experience with these techniques comes from experiments with a prototype system called Draco [Neighbors84a]. Referring to figure 9 we have been fortunate to have a few people try their hands as “Application Domain Analysts.” Some of the results have been published [Gonzalez81, Sundfor83a, Sundfor83b]. Mostly application domains have been created by people from industry interested in a particular application problem. Although these experiences have only occasionally resulted in working small application programs[3] I can say that the technique has been a success. The technique has been a success because in every case the analyst has come away from the domain analysis process with an improved understanding of the parts of a system that make up a solution system in the problem domain. This improved understanding comes from considering the problem independent of implementation or architecture. More recent work [Dunn91] proves the power of application domain analysis to describe classes of systems.

We have been less successful in interesting people to try being a “Modeling Domain Analyst.” In my dissertation [Neighbors80] I tried out the idea of modeling domains using Draco. The idea appeared to work and brought out a lot of interesting issues about maintaining consistent implementations during refinement. Draco was used as a mechanism to convert itself from one computer to another [Arango86]. This was a translation from one level of abstraction to the same level of abstraction. The idea of “modeling domains” was not severely tested. The concept of modeling domains is strongly supported by the work of Batory [Batory88]. This work describes a hierarchy of modeling domains that supports the construction of database systems with widely different features. Considering that a database system has been the core of every large system I have examined this is clearly a complex and important set of modeling domains. As with application domains I would think our experience with modeling domains a success because in every case the modeling domain analyst has come away from the experience with a better understanding of the domain. For modeling domains this improved understanding comes from considering architectures and implementations without expanding the function of the domain.

We have done a lot of work on the models and mechanisms for constructing systems using transformational implementation methods. The Draco methodology [Neighbors80] is an instance of such a method. Prieto-Diaz [Prieto-Diaz85] studied organizational schemes for libraries of software components. This technique has an immediate payoff for organizations and is a good place to start. Ultimately the library problems discussed earlier will limit this approach. Arango [Arango88] developed a model for classifying and discussing methods of this kind. These models make clear what kind of knowledge these types of methods use and how it is used. Baxter [Baxter90] studied the problem of re-implementing a particular system developed under this method if the system specification changes. No method of this type will be a success without a solution to this problem because the system specification will change. Finally, Srinivas [Srinivas90] considered methods for capturing the description of a domain as a formal algebraic theory. The rigor of a formal method would certainly be welcome over the informal way we combine domains now. However, it must avoid the notational problems and computational complexities that prohibited previous formal methods from succeeding.

A message of this work is that neither sophisticated Artificial Intelligence planning mechanisms nor formal theory proof mechanisms are required to improve the productivity of programmers. Often we have consciously avoided the use of such mechanisms to reduce the burden on the tool user who just wants a working program out. Program generators are the extreme example of this. We cannot expect the tool users to give advice to complex AI planning mechanisms they did not create or provide statements in a formal algebraic theory they did not produce. The Draco Methodology is an extremist knowledge-based approach. It consciously trades domain-specific knowledge against powerful general mechanisms. Simple mechanisms and encapsulated domains will allow the use of higher-level sophisticated planning techniques to refine specific problems once a critical mass of problem and modeling domains is available.

Lessons from Using a Prototype System:

1. Programs refined this way are very efficient. Optimizing transformations applied at a level of abstraction above common programming languages are the key. These are seldom discussed in the literature because the abstractions (through domain analysis) are hard to determine and not usually of general interest.
2. After doing many examples consisting of the application of thousands of optimizations and component refinements it becomes clear that the ability to use subsystems consisting of pre-optimized and pre-refined parts of existing domain hierarchies is important. For large modeling domains such as database concepts it is important that a system-specific implementation *can* be refined by the system. However, most of the time you would not *want* to refine the default, general version of a database in detail.
3. Academics are generalists. As generalists they prefer to work on the general part of the problem, the refinement mechanism. They are not really motivated to produce application problem domains that test their mechanisms.
4. Industry causes people to specialize. As specialists they prefer to work on the domain specific part of the problem, the application domains. They are not really motivated to change the mechanism. The mechanism must be understandable and produce real software.
5. No one wants to make modeling domains. For industry, modeling domains do not directly apply to the problem at hand. For academia, producing a modeling domain does not add any “new” knowledge. It simply structures what we already know.

Academic and industry cooperation is clearly required to produce useful application domains that rely on strong, general modeling domains.

3. By small programs we mean 2000 to 5000 source line programs. These are toy programs with respect to the large systems at which these techniques are aimed.

My experience is that the structuring of what we already know has produced some of the very best work in Computer Science. The process of structuring the knowledge points out what we do not know. Compilers and operating systems are old examples of this process. These areas started as collections of ad hoc engineering artifacts. They currently stand as general architectures backed by formal theory and described in textbooks. This is the evolutionary path for a modeling domain. Producing a general architecture and related formal theory for areas that are less evolved, such as databases and networks, is a difficult task.

Conclusions

The concept of "Domain Analysis" has been embraced by many for quite a few different reasons [Prieto-Diaz91]. Domain analysis results provide an organization with the following capabilities:

- Use the domain model to check the specifications and requirements for a new required system in the domain.
- Educate people in the organization providing them with the general structure and operation of systems in the domain.
- Derive working systems directly from the statement of the system in domain specific terms.

I concentrate exclusively on the derivation of working systems. I have completely ignored the other uses. I see the effects, but I do not focus on them. Other research groups are beginning to investigate the important educational and quality aspects of domain analysis [Prieto-Diaz91]. I am glad that the knowledge of the "old hand" Domain Analysts has been acknowledged.

Currently there are large military and industrial research efforts specifically aimed at application domain analysis. These will have to deal with the lack of modeling domains or turn into simple program generators. There are large academic research efforts to develop new mechanisms for refinement and transformation. These will have to deal with planning using a complex mechanism and computational complexity.

We have tested a simple method for software construction using components that is derived from the literature and industry experience. It works. Using this method McIlroy's software component factory problem turns into a domain hierarchy construction problem. This is a much harder problem. Early enthusiasm came from simple operations (sorts, transcendentals) on simple objects (numbers). These are not the big system problem. The big system problem is refining big complex systems that deal with big modeling domains like databases, operating systems, communications, and graphics. It is imperative that work which forms modeling domains from the existing Computer Science literature and practice be recognized as an important contribution. Without strong modeling domains the vision of software construction using components will go no further.

References

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

[Arango88]
G. Arango, *Domain Engineering for Software Reuse*, Ph.D. Dissertation, ICS Dept., University of California at Irvine, 1988.

[Balzer73]
R. Balzer, "A Global View of Automatic Programming", *3rd International Joint Conference on Artificial Intelligence*, pp. 494-499, SRI International, August 1973.

[Batory88]
D. Batory, "GENESIS: An Extensible Database Management System", *IEEE Transactions on Software Engineering*, vol. SE-14, no. 11, pp. 1711-1730, November 1988.

[Baxter90]
I. Baxter, *Transformational Maintenance by Reuse of Design Histories*, Ph.D. Dissertation, ICS Dept., University of California at Irvine, 1990.

[Booch87]
G. Booch, *Software Components with Ada*, Benjamin/Cummings, 1987.

[Campos78]
I. Campos and G. Estrin, "Concurrent Software System Design Supported by SARA at the Age of One", *3rd International Conference on Software Engineering*, pp. 230-242, IEEE, May 1978.

[Cooprider79]
L. Cooprider, *The Representation of Families of Software Systems*, Ph.D. Dissertation, Carnegie-Mellon University, Computer Science Dept., CMU-CS-79-116, April 1979.

[Corwin72]
W. Corwin and W. Wulf, *A Software Laboratory Preliminary Report*, Report CMU-CS-71-104, Carnegie-Mellon University, August 1971.

[Darlington73]
J. Darlington, "A System Which Automatically Improves Programs", *3rd International Joint Conference on Artificial Intelligence*, pp. 479-485, SRI International, 1973.

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

[Dijkstra68]
E. Dijkstra, "Complexity Controlled by Hierarchical Ordering of Function and Variability", in *Software Engineering*, P. Naur and B. Randell eds., NATO Science Committee Report, pp. 181-185, Germany, October 1968.

[Dijkstra69]
E. Dijkstra, "Structured Programming", in *Software Engineering Techniques*, J. Buxton and B. Randell eds.,

NATO Science Committee Report, pp. 84-88, Italy, October 1969.

[Dunn91]
M. Dunn and J. Knight, "Software Reuse in an Industrial Setting: A Case Study", *13th International Conference on Software Engineering*, pp. 329-338, May 1991.

[Feldman72]
J. Feldman, *Automatic Programming*, Report STAN-CS-72-255, Stanford University, February 1972.

[Fickas85]
S. Fickas "Automating the Transformational Development of Software," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1268-1277, November 1985.

[Freeman87]
P. Freeman, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no.7, pp.830-844, July 1987.

[Green69]
C. Green, "Application of Theorem Proving to Problem Solving", *1st International Joint Conference on Artificial Intelligence*, pp. 219-239, University Microfilms 1969.

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

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

[Knuth74]
D. Knuth, "Structured Programming with GOTO Statements", *ACM Computing Surveys*, vol. 6, no. 4, pp. 261-3-1, December 1974.

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

[McIlroy68]
D. McIlroy, "Mass Produced Software Components", in *Software Engineering*, P. Naur and B. Randell eds., NATO Science Committee Report, pp. 138-155, Germany, October 1968.

[Neighbors80]
J. Neighbors, *Software Construction Using Components*, Ph.D. Dissertation and Report UCI-ICS-TR160, University of California, Irvine, ICS Dept., 1980.

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

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

[Neighbors89]
J. Neighbors, "Draco: A Method for Engineering Reusable Software Systems", in *Software Reusability*, T. Biggerstaff and A. Perlis eds., vol. 1, pp. 295-319, Addison-Wesley 1989.

[Parnas72]
D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM*, vol. 15, no. 12, December 1971, pp. 1053-1058.

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

[Prieto-Diaz85]
R. Prieto-Diaz, *A Software Classification Scheme*, Ph.D. Dissertation, ICS Dept., University of California at Irvine, 1985.

[Prieto-Diaz86]
R. Prieto-Diaz and J. Neighbors, "Module Interconnection Languages", *The Journal of Systems and Software*, vol. 6, pp. 307-334, November 1986.

[Prieto-Diaz87]
R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, pp. 6-16, January 1987.

[Prieto-Diaz91]
R. Prieto-Diaz and G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Press, 1991.

[Sammet76]
J. Sammet, "Programming Languages", *Encyclopedia of Computer Science*, pp. 1169-1174, Petrocelli/Charter 1976.

[Schorre64]
D. Schorre, "META II: A Syntax-Oriented Compiler Writing Language", *Proceedings of the ACM National Conference*, pp. D1.3-1 to D1.3-11, ACM 1964.

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

[Srinivas90]
Y. Srinivas, *Algebraic Specification: Syntax, Semantics, Structure*, Report 90-15, ICS Dept., University of California at Irvine, 1990.

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

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

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

[Tichy79]
W. Tichy, "Software Development Control Based on Module Interconnection", *4th International Conference on Software Engineering*, pp. 29-41, September 1979.

[Wirth71]
N. Wirth, "Program Development by Stepwise Refinement", *Communications of the ACM*, vol. 14, no. 4, April 1971, pp. 221-227.