THE DRACO APPROACH TO CONSTRUCTING SOFTWARE FROM REUSABLE COMPONENTS

James M. Neighbors

Reuse Project
Information and Computer Science Department
University of California
Irvine, CA 92717

ABSTRACT

This paper discusses a mechanism called Draco which aids in the construction of software systems. In particular we are concerned with the reuse of analysis and design information in addition to programming language code.

The goal of the work on Draco has been to increase the productivity of software specialists in the construction of *similar* systems. The particular approach we have taken is to investigate the construction of software from reusable software components which are organized by problem domain. The experimental method used was to hypothesize a scheme based on previous work and experiment with example problems on a prototype system.

INTRODUCTION

It has been a common practice to name new computer languages after stars. Since the system described in this paper is a mechanism which manipulates special purpose langauges it seems only fitting to name it after a structure of stars, a galaxy. Draco is a dwarf elliptical galaxy in our local group whose small size and close distance to home is well suited to the current system which is a small prototype.

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

What Does Draco Do?

There are basically only three activities performed by Draco:

1. Draco accepts a definition of a problem domain as a high-level domain-specific language which we call a *domain language*. Both the syntax and semantics of the domain language must be described.
2. Once a domain language has been described, Draco can accept a description of a software system to be constructed as a statement or program in the domain language.
3. Once a complete domain language program has been given then Draco can refine the statement into an executable program under human guidance.

While the above three tasks are easily stated they are difficult to achieve and each will be described in detail in a following section. The fine details, relevant work, and design tradeoffs of the Draco approach are discussed in depth by Neighbors (Neighbors, 1980a). The features and limitations of the current Draco implementation are given in the Draco manual (Neighbors, 1980b).

THE ORGANIZATIONAL CONTEXT OF DRACO

Before we go further into the technical details of a Draco specification and refinement it is helpful to understand who supplies these details. Figure 1 shows the flow of information between people in different

roles external to Draco.

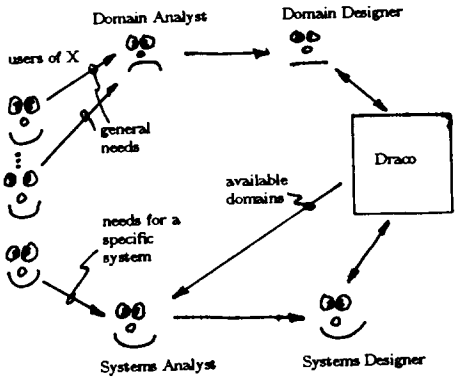

Figure 1. Organizational Context of Draco

Classically 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 two new major human roles: the *domain analyst* and the *domain designer*. The 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 usually only sucessfully done by a person who has built many systems for different customers in the same problem area. Once the domain analyst has described the objects and operations which are germane to the 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.

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 orgainzation'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 the required system then the system 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 program 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. In addition, Draco will produce different implementations with different execution speed and space characteristics from the same problem description. The Draco response to rapid prototyping is to produce implementations using very safe and simple implementation structures. In this context experience has shown that refinement proceeds with very little human guidance.

WHAT COMPRISES A DOMAIN DESCRIPTION

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

1. **Parser** The parser describes the external syntax of the domain and the *prefix internal form* of the domain. The syntax 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. It is *not* a parse tree. The internal form is the data actually manipulated by Draco. Some consistency checks may be performed by the parser such as the data flow consistency that all data produced is consumed and all data consumed is produced.
2. **Prettyprinter** The prettyprinter description tells Draco how to produce the external syntax of the domain for all possible program *fragments* in the domain. This is necessary because the mechanism must be able to interact

with people in the language of the domain and discuss incomplete parts of the problem.

3. **Transformations** The program transformations on a domain language program are *strictly* source-to-source transformations on the objects and operations of the domain. These transformations represent the rules of exchange between the objects and operations of the domain and are guaranteed to be correct independent of any particular implementation chosen for any object or operation in the domain. This is an important concept and different from the usual definition of program transformations. The concept of transformation used in the USC/ISI Transformational Implementation system (Balzer, Goldman, and Wile, 1976) and the Harvard Program Development System (Cheatham, Holloway, and Townley, 1979) is a combination of Draco transformations *and* refinements. Draco program transformations *never* make implementation decisions they only represent optimizations *at the domain language level.*

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.* A refinement at a high-level of abstraction is similar to the concept of a plan in the MIT Programmer's Apprentice system (Waters, 1982). The semantics of a component must of course be recursive to allow such representations as lists as arrays and arrays as lists. In addition, the refinements must specify their implementation decisions explicitly so that Draco can maintain the consistency of the developing program. These decisions are specified, kept and used in a form similar to that used in Module Interconnection Languages (MILs) which are discussed in (Prieto-Diaz and Neighbors, 1982).

5. **Procedures** Domain-specific procedures are used in circumstances where the knowledge to do a certain domain-specific transformation is algorithmic in nature. An example is the construction of LR(k) parser tables from a grammar description. These procedures are similar to the Draco transformations in that they only operate in one domain and never reach across domain boundaries to make implementation decisions.

Thus, the basis of the Draco work is the use of domain analysis to produce domain languages which may be transformed for optimization purposes and implemented by software components each of which contains multiple refinements which make implementation decisions by restating the problem in other Draco domain languages. Each of these domain parts will be discussed in more detail in a dedicated section below.

Figure 2 shows how some hypothetical domains might be connected together to build a "statistics reporting domain".

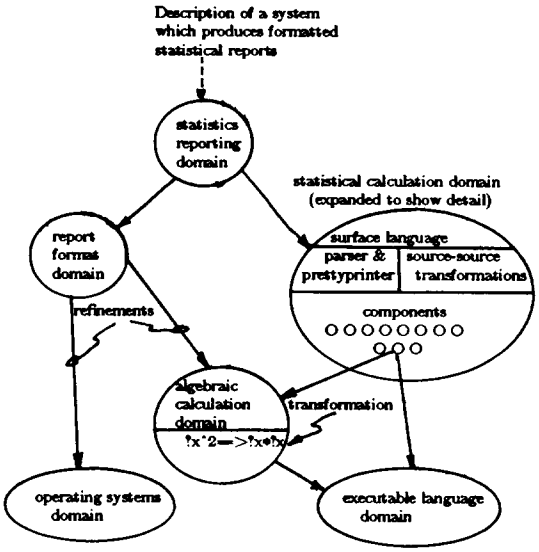

Figure 2. An Example Domain Organization

It is important to keep in mind that the "report format domain" might be used by many other domains other than the "statistics reporting domain" and this too is the reuse of analysis. It has been our experience that some domains such as a database domain are used in the design of many other

domains. Thus in our figure we are not just discussing a reusable "statistics reporting domain" we are really discussing the reusable domains of "report formatting" and "statistical calculation". The "statistics reporting domain" is simply an instance of reuse.

While some domains sound very problem specific such as an "Air Defense System" domain where there is not a lot of demand for many systems, keep in mind that these domains are primarily built out of problem inspecific and much more reusable domains (e.g., database domain, graphics domain) which are tailored in the refinement process to the specific problem.

The language scheme is neither *wide-spectrum* nor *narrow-spectrum*. In a wide-spectrum scheme one language suffices to represent the developing program from specification to final executable code. In a narrow-spectrum approach the problem progresses through a strict hierarchy of languages (e.g, specification language, design language, executable language.) Draco is a scheme where the problem is in many languages at once where the languages are not in a strict hierarchy.

SOFTWARE COMPONENTS AND THE REFINEMENT PROCESS

There is one software component for each object and operation in a domain and each software component consists of multiple refinements. Each refinement makes an implementation decision by restating the semantics of the object or operation in terms of other domains know to Draco. The *refinement process* is the process of restating the problem originally specified in one domain repetitively into other domains. During this process the consistency of the developing program must be maintained and its level of abstraction must eventually decrease to an executable (or compilable) language.

An example software component for exponentiation in a low-level language is shown in figure 3. We must emphasize that this does *not* represent a high-level language and while Draco has domains which represent this level of general-purpose languages our aspirations and experience are with much more problem-specific domains. The necessity of an example in a commonly understood domain forces us to this level.

```
COMPONENT: EXP(A,B)
PURPOSE:  exponentiation, raise A to the Bth power
IOSPEC:   A:number,B:number/number
DECISION: The binary shift method is O(ln2(B)) and requires
          B integer=>0 while the Taylor expansion is an
          adjustable number of terms and requires A>0.

REFINEMENT: binary shift method
CONDITIONS: A a SIMAL number
          B a SIMAL integer
BACKGROUND: Knuth's SemiNumerical, Algorithm A, pg. 399
INSTANTIATION: FUNCTION, INLINE
CODE: SIMAL.BLOCK
    [[ IF B<0 THEN EXCEPTION ;
       POWER:=B ; NUMBER:=A ; ANSWER:=1 ;
       WHILE POWER>0 DO
         [[ IF POWER&1=1 THEN ANSWER:=ANSWER*NUMBER ;
            POWER:=POWER>>1 ;
            NUMBER:=NUMBER*NUMBER ]] ;
       RETURN ANSWER ]]
END REFINEMENT

REFINEMENT: Taylor expansion
CONDITIONS: A,B as SIMAL numbers
BACKGROUND: VNR Math Encyclopedia, pg. 490
INSTANTIATION: FUNCTION, INLINE
ADJUSTMENTS: TERMS[20] - number of terms, error is
          approximately (B*ln(A))^TERMS/TERMS!
CODE: SIMAL.BLOCK
    [[ IF A<=0 THEN EXCEPTION ;
       SUM:=1 ; TOP:=B*LN(A) ; TERM:=1 ;
       FOR I:=1 TO TERMS DO
         [[ TERM:=(TOP/I)*TERM ;
            SUM:=SUM+TERM ]] ;
       RETURN SUM ]]
END REFINEMENT
END COMPONENT
```

Figure 3. An Example Software Component

The example component is has two refinements which refine the exponentiation operator in the SIMAL domain† back into the SIMAL domain by making implementation decisions. Notice that the choice of one or the other of these refinements will cause the final programs to act differently even if all other implementation decisions are the same. The exception conditions for these two methods are different as are the acceptable types and ranges.

During the refinement process a *systems designer* would be responsible for making two different decisions with respect to this component. First it must be decided which *refinement* to use and second it must be decided what kind of *functional structure* will result from the refinement.

The decision of which refinement to use must be made unless some of the refinements have been excluded by ASSERTIONS in previously used refinements. As an example, if it had previously been decided that all numbers in the domain would be represented in float-

†SIMAL is a simple algorithmic language.

ing point notation then the "binary shift method" would be automatically excluded from use by the Draco consistency check mechanism. ASSERTIONS specify implementation decisions if a refinement is used and CONDITIONS specify decisions which must have been made before a refinement may be used. Primarily the domain object refinements make ASSERTIONS and domain operation refinements check CONDITIONS.

Once a refinement for a component is chosen the systems designer must describe how the refinement is to fit into the structure of the developing program. The different possibilities are governed by the INSTANTIATION field of the refinement. By structure we mean the function-procedure call structure of the resulting program as might be shown in a Structured Design structure chart. If the refinement is used INLINE then one can think of the refinement mechanism as simply expanding the refinement in the developing program as a macro.* If the refinement used as a FUNCTION then the body of the refinement is set aside as a function definition and the original use of the component is replaced with a call to the function.‡ Between these two extremes is PARTIAL instantiation where some of the arguments are instantiated inline and some are passed.§

These decisions must be considered by the systems designer for each and every component in every domain in the developing system. These are far too many decisions for a person to make or even consider for a large system. The refinement mechanism in Draco provides three primary mechanisms for dealing with this complexity: domains, locales, and tactics.

Domains as an Aid to Refinement Complexity

Aside from the convenient encapsulation of a problem area that the concept of domains provides to the domain and systems analysts, the domain concept is also of use to the designers. The original problem is stated in the domain language of a single domain but the instant the refinement process is invoked on the problem it ends up as a statement in many domains at once! These domains are being used as *modelling domains* for the problem (Tonge and Rowe, 1975). Figure 4 graphically illustrates the refinement process from a statement in only the problem domain, through many modelling domains, into the final target domain.

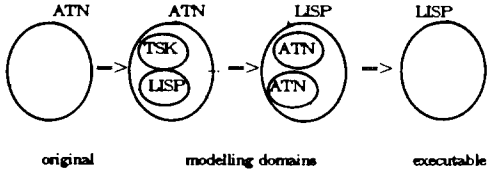

Figure 4. Domains in the Refinement Process

The systems designer works with the refinement mechanism in one domain at a time. In a developing program there may be multiple *instances* of a domain, the refinement mechanism may be directed to scan all instances during refinement or 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.)

Locales Aid in the Refinement of Efficient Programs

Within an instance of a domain a systems designer may limit the refinement mechanism to a *locale* within that instance. Typically the locale would be tightened about an "inner loop" of the program which has been determined from a previous refinement. Here the systems designer would want to go through the available refinements very carefully and choose more elaborate and efficient implementations. Once these decisions have been made the locale may be enlarged up to the size of the original instance and the refinement mechanism consistency checker will make sure that all implementations chosen are consistent with the very fancy implementations in the inner loops.

Tactics for Refinement

The domain and locale concepts limit the scope of what the systems designer has to

*With variable renaming and access problems it is a bit more complex than a macro expansion but this is a useful model of the process.

‡The types and accesses to the passed parameters must be right of course.

§In the PARTIAL case Draco must keep track of different versions of the functions.

think about. The refinement tactics limit the sheer number of decisions the systems designer must make.

The tactics are domain independent rules for making refinement decisions. They are not guaranteed to make a decision and when they don't the systems designer must make the decision. Tactics must obey the refinement consistency checker just as the systems designer must. Some example tactics are summarized below:

1. If we've already defined a function which implements this component then use that function.
2. If there is a refinement to this component which can be made into a function then use it as a function.
3. Use the default refinement for the component (the first one specified) with the default instantiation (the first instantiation specified.)
4. Use the refinement with the minimum number of assertions and conditions.

The Draco approach to rapid prototyping is to build tactics which use the default refinement under the default instantiation. These refinements are usually very general (i.e., few conditions and assertions) but expensive in time and space. Very little interaction is required from the systems designer in this case other than selecting the domains and using the rapid prototyping tactics.

We refer to this mechanism as tactics because policy decisions are made without respect to the global context and Draco domain organizational structure. We expect to investigate refinement strategies and will discuss these later.

Recording the Refinement Process

The refinement process does *not* proceed strictly top-down or from one language level to another. In fact, sometimes it is necessary to back-up 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 program. There are two uses for this refinement history: to understand the resulting program and to guide the *refinement replay* of the problem if the original specification is changed and a new implementation is needed (Wile, 1982). We have found that the refinement history is much larger than the resulting program code.† This large amount of information is lost to someone attempting to reuse an existing piece of source code.

In general we have found the making of decisions to proceed as shown in figure 5.

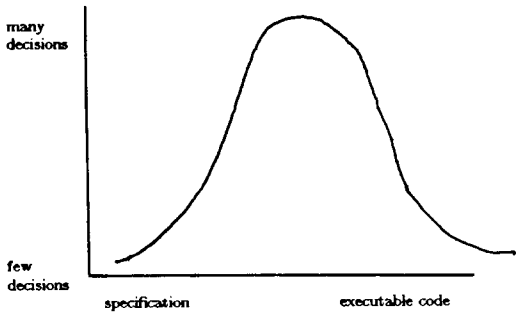

Figure 5. Decisions Pending versus Abstraction Level

The number of decisions to be made rises initially as implementation decisions must be made in the modelling domains and decreases finally as the already made decisions constrain the remaining decisions to only one choice. The *intermediate modelling swell* of this graph is the largest barrier to refinement.

DOMAIN SPECIFIC SOURCE-TO-SOURCE PROGRAM TRANSFORMATIONS

It should be possible to refine a statement in any domain without the use of program transformations. The transformations which state rules of exchange between statements in a domain language and statements in that same domain language (i.e., source-to-source transformations.)

The primary use of the transformations is to optimize a domain language program. In general, the transformations are simple enough that they are seldom of use on a domain language program written by systems designer. The transformations are usually obvious statements. The following might be transformations in a simple low-level language.

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

```
ADDx0:  ?x+0 => ?x
MPYx0:  ?x*0 => 0
IFTRUE: IF TRUE THEN ?s1 ELSE ?s2 => ?s1
```

However, when a very general refinement of a component is used in a specific context the Left-Hand-Sides (LHS) of these simple rules of exchange tend to appear. Thus, the program transformations specialize the refinements of components to their use in a specific system at levels of abstraction far above that of executable code.

The transformations do not make or use implementation decisions but they do change the execution of the program. Certain side-effects must be checked or prohibited. The type and checking of these enabling conditions is discussed in (Standish et. al., 1976). As an example, in the transformation MPYx0 above the fact that the code fragment ?x does not write any data as a side-effect must be checked. These effects are usually not checked in Draco because we attempt to build languages to prohibit the side-effect style.

The Transformation Naming Problem

A domain may easily have more than two thousand transformations and we do not believe having the systems designer remember all the names or look them up in a catalog would be very successful. In the early program transformation systems the user would decide where to apply each transformation and which transformation to apply. This would be chaos in a Draco development with large systems and many domains.

In Draco all program fragments held in the internal form are annotated with all the program transformations which could apply. This includes domain language programs which have been parsed, refinement code bodies, and the RHS of all transformations. Thus, the systems designer never suggests transformations but instead *solicits suggestions* from Draco as to which transformations can be applied to a given program fragment.

As some transformations are applied they suggest still other transformations which could apply. The scheme which keeps this process going is the interpretation of *transformation metarules*. The metarules relate the transformations of a domain to other transformations in that domain (Kibler, Neighbors, and Standish, 1977). They are calculated as transformations are added to the transformation library of a domain. As an example metarule, the RHS of the transformation MPYx0 would contain a metarule which would suggest the transformation ADDx0 at a higher context because the LHS of ADDx0 contains a 0 and MPYx0 introduces a 0 into the program.

Controlling the Transformation Mechanism

If there are symmetric transformations in the transformation set (and this is usually the case) the the transformation mechanism must be carefully controlled. Two low-level symmetric transformations are:

```
(IF ?p THEN ?s1 ELSE ?s2)*?x => (IF ?p THEN ?s1*?x ELSE ?s2*?x)

(IF ?p THEN ?s1*?x ELSE ?s2*?x) => (IF ?p THEN ?s1 ELSE ?s2)
```

The transformation mechanism in general only applies its transformations to the domain locale selected by the systems designer. Each transformation has an application code (a number between 00 and 99) and these are used to form groups. The different groups are:

100 and up	Markov Algorithm actions
90-99	simplifying transformations
80-89	canonical form
60-79	operator arrangement
40-59	flow statement arrangement
20-39	program segment arrangement
10-19	reverse canonical form
00-09	Markov Algorithm invocations

Thus in the case of symmetric transformations they must not both be in the same group. The transformations suggested in a particular place in the program are tried in a strict highest application code first order. The systems designer performs transformations by specifying the application code range or the group name for the transformations in the locale to be tried.

The Markov algorithms mentioned are a scheme for producing procedural results such as data-flow analysis and type propagation from a cooperating set of source-to-source transformations with metarule interpretation.

The Importance of Performing Source-to-Source Transformations at the Right

Level of Abstraction

Assume that we have the transformation†

EXPx2: ?x^2 ==> ?x*?x

in the SIMAL language which converts an exponentiation to a multiply in a special case. This transformation is actually a manipulation of the exponentiation component presented in figure 3. Once again we must apologize for using such a low-level language with the goal of having a common language of discourse. The concepts we will discuss apply equally to the much higher-level, less-general domain languages.

Given the above transformation and the Taylor expansion refinement from the EXP component in figure 3, then figure 6 presents the possible actions to refine the SIMAL exponentiation operator into LISP with no exponentiation operator.‡

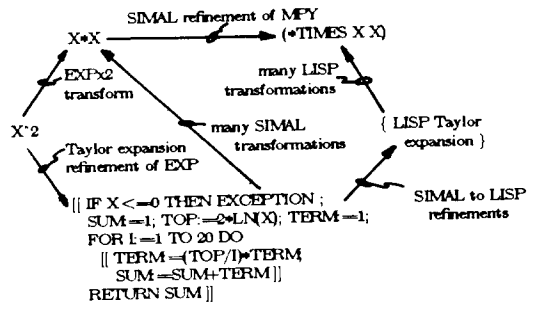

Figure 6. Possible Actions on a SIMAL EXP

Once the Taylor expansion has been used no set of equivalence preserving program transformations can achieve the same effect as the EXPx2 transformation without *recognizing* that a Taylor expansion is the mechanism being used. Such transformations would be far too specific to be useful. The problem is far more aggravated when we map into a language such as LISP which we are assuming has no exponentiation operator.‡ The designer of the LISP domain would not include transformations to recognize and deal with a Taylor expansion. Clearly this example is extreme in that we used an *approximation,* the Taylor expansion, to prove the point. If we had used the "binary shift method" of figure 3 many powerful SIMAL transformations would have been required to manipulate the results of the refinement into a simple multiply as EXPx2 does.

In our experiments with Draco we constructed an Augmented Transition Network (ATN) domain. An ATN is a class 0 parser which runs in parallel execution to do natural language parsing. We have had ATN transformations remove states (the equivalent of parse rules) from parser descriptions. Extremely powerful transformations would have no chance of removing these same states from the resulting parallel executing LISP program.

The point here is that truly powerful optimization can be achieved with fairly simple mechanisms at the correct level of abstraction (i.e., at the domain language level.) These optimizations are far more powerful than all of the classical optimizations (e.g., code motion, register allocation, reuse of temp space, data flow analysis, reduction in operator strength, etc.) which are applied to general-purpose low-level languages. *The use of source-to-source transformations on domain languages is the key to building* **efficient** *programs out of reusable and general software components.*

WHAT CAN BE SAID ABOUT THE POWER OF THE DRACO METHOD?

Given a collection of Draco domains with their interdependencies and their CONDITIONS and ASSERTIONS we would like to know what we can do with them. In particular, given a specific domain language program, a set of domains, and a target domain (an executable language), we would like to be able to answer the following *reusability questions.*

1. Can we refine the domain language program completely into the target domain?
2. If so, can Draco provide an implementation for the given domain language program?
3. If not, what extra implementation knowledge is needed?

†EXPx2 requires the enabling condition that ?x be side-effect-free.

‡Most LISPs do have an exponentiation operator, however, we are attempting an analogy between high-level languages using low-level languages.

A Petri net model of the implementation decisions in the Draco domains can be constructed and *the reusability questions can be treated as the Petri net reachability problem.*

In particular, we were surprised to find that reusability questions 1 and 2 are *decidable!* It is unknown whether reusability question 3 is decidable or not.

Since reusability questions 1 and 2 are decidable, then the problem of *refinement deadlock* is decidable. The refinement deadlock problem plagues system designers and it occurs when the designer makes an implementation choice which will later on prohibit the program from being refinable into a given target domain. A program cannot be refined when no consistent implementation exists with respect to the implementation decisions already made and implementation choices known to Draco. After each decision the systems designer would like to know if a complete refinement of the problem into the target domain still exists. Refinement deadlock primarily occurs when the designer is using elaborate refinements with many CONDITIONS and ASSERTIONS.

The complexity of the Petri net reachability problem prevents the straightforward use of the results. The complexity of answering the reusability questions for a medium-sized augmented transition network refined into only one other domain is approximately O(2^100). We don't regard this terribly high complexity as a real problem because in using full Petri nets we are using far too general a model. We suspect that a less general model and AI planning techniques will suffice as a basis for refinement strategies which are directly aimed at the reusability questions.

CONCLUSIONS

The Draco 1.0 system† became operational in 1979 (Neighbors, 1980a and 1980b). We have found experimenting with a complete prototype rewarding. The details and results of these experiments have been reported elsewhere (Neighbors 1980a, Gonzales 1981, Sundfor 1983a and 1983b). Our results and experience with the method is summarized in the following sections.

Some *real* Draco domain language programs are given in the appendix. The concepts presented in this paper using low-level language examples are really applied using Draco to these high-level languages.

Reuse of Analysis, Design, and Code

We have found that the reuse of analysis and design is much more powerful than the reuse of code. We realize that in the *short-term* the reuse of code has the largest payoff and software producing organizations should start to exploit this technology now (Freeman, 1983). Code is very tricky to reuse. Many of the analysis, design, and implementation decisions which went into its construction are absent from the code itself. Our experience of refinement histories more than 10 times the size of the final code speaks for the large amount of information missing in just the code itself.

Domain Analysis and Design

Only about 10 or 12 full usable Draco domains have been built and each has reenforced the idea that domain analysis and design is **very hard**. Typically it takes someone who is an *expert* in a particular problem area four months to just start to show progress with the domain. Even with well-documented work such as the ATNs it takes some time to read all the literature and extract an appropriate set of objects and operations. It takes further time to find an appropriate syntax for these structures. This is creative work and similar in scope to writing a book or survey paper on the area. As an example, the domain analysis of tactical display systems given by Sundfor (Sundfor, 1983a) is over 100 pages long.

One sure way to make the Draco method fail is to take unexperienced personnel and have them observe the "old hands" work to construct a domain. Its easy to construct a bad domain and very hard to construct a good one. The reasons are similar to the reasons for the failure of extensible languages (Standish, 1971).

†We expect the next major version, Draco 2.0, to be available in August 1983 at which time the emphasis will switch to the instrumental use of Draco.

Reusable Software and Efficiency

It is not true that software constructed from reusable software components is inefficient.
We have shown that using the Draco method systems with different implementations and different modular structures can be created. Each of these has a different time-space execution characteristic.

In addition we discussed the use of domain-specific transformations to provide optimization at the correct level of abstraction which results in more powerful optimizations than usually available to users of general-purpose languages.

The Draco Method

The Draco method described here provides a context where program transformations, module interconnection languages, software components, and domain-specific languages work together well in the construction of software.

From experience with Draco we submit that usable software can be constructed from reusable software components.

BIOGRAPHY

Jim Neighbors is working with new Software Engineering techniques to eliminate redundancy in designing and developing software systems that are similar in nature even though they address different applications. He has proposed and demonstrated an approach to the reuse of analysis and design results that incorporates several interesting concepts. These include high-level specification languages, semi-automatic production of executable code and domain analysis which is a generalized form of symstems analysis that captures the essence of an entire class of applications in the form of high-level specification languages. Current work is focused in extending this technology to permit the generation of production-quality systems. Dr. Neighbors received his B.S. in Computer Science, and his B.A. in Physics in 1974 and his Ph.D. in 1980 from the University of California, Irvine.

Mailing address: Information and Computer Science Department, University of California, Irvine, CA, 92717.

Telephone: 714-856-7403.

REFERENCES

Balzer, R., Goldman, N. and Wile, D.On the Transformational Implementation Approach to Programming. In the proceedings of the 2nd International Conference on Software Engineering. IEEE, 1976, 337-344.

Burton, R. Semantic Grammar: A Technique for Efficient Language Understanding in a Limited Domain (Ph.D. Thesis). Irvine, CA: University of California, ICS Dept., 1976

Cheatham, T.E., Holloway, G.H., and Townley, J.A. Program Refinement by Transformation. In the proceedings of the 5th International Conference on Software Engineering. IEEE, 1981, 430-437.

Freeman, Peter Reusable Software Engineering: Concepts and Research Directions. Newport, RI: ITT, 1983.

Gonzalez, L. A Domain Language for Processing Standardized Tests (MS Thesis). Irvine, CA: University of California, ICS Dept., 1981.

Kibler, D., Neighbors, J.M., and Standish, T.A. Program Manipulation via an Efficient Production System. SIGPLAN Notices, 1977, 12(8), 163-173.

Neighbors, J. Software Construction Using Components (Ph.D. Thesis and Tech. Rep. TR-160). Irvine, CA: University of California, ICS Dept., 1980a.

Neighbors, J. Draco 1.1 Manual (Tech. Rep. TR-156). Irvine, CA: University of California, ICS Dept., 1980b.

Prieto-Diaz, R. and Neighbors, J. Module Interconnection Languages: A Survey (Tech. Rep. TR-189). Irvine, CA: University of California, ICS Dept., 1982.

Standish, T.A. PPL - An Extensible Language That Failed (Report 15-71). Cambridge, MA: Harvard University, Center for Research in Computing

Technology, 1971.

Standish, T.A., Harriman, D.C., Kibler, D.F. and Neighbors, J.M. The Irvine Program Transformation Catalogue (Tech. Rep.). Irvine, CA: University of California, 1976.

Sundfor, S. Draco Domain Analysis for a Real Time Application: The Analysis (Tech. Rep. RTF 015). Irvine, CA: University of California, 1983a.

Sundfor, S. Draco Domain Analysis for a Real Time Application: Discussion of the Results (Tech. Rep. RTP 016). Irvine, CA: University of California, 1983b.

Tonge, F. and Rowe, L. Data Representation and Synthesis (Tech. Rep. TR-63). Irvine, CA: University of California, ICS Dept., 1975.

Waters, R.C. The Programmer's Apprentice: Knowledge Based Program Editing. IEEE Transactions on Software Engineering, 1982, 8(1), 1-12.

Wile, D.S. Program Developments: Formal Explanations of Implementations. (Res. Rep. ISI/RR-82-99). Marina Del Rey, CA: Information Sciences Institute, 1982.

Woods, W.A. Transition Network Grammars for Natural Language Analysis. Communications of the ACM, 1970, 13(10),591-606.

APPENDIX

In the body of the paper we discuss Draco's manipulation of high-level languages in terms of analogies with low-level languages like SIMAL. This appendix presents program fragments from actual high-level Draco domains.

Augmented Transition Network Example

The first example from (Neighbors, 1980a) presents an example the augmented transition network domain.

```
ATN WOODS
NETWORK SENTENCE
; Example ATN from both (Woods, 1970) and (Burton, 1976)
; Abbreviations
;   NP      =       noun phrase             NPR      = nomitive pronoun
;   PPRT    =       past participle         ADJ      = adjective
;   ITRANS  =       intransitive            AGNTFLG  = agent possible
;   TRANS   =       transitive              DET      = determiner
;   PREP    =       preposition             S-TRANS  = sentence object
;   PRO     =       pronoun                 AUXVERB  = auxiliary verb
;
;from    to    |    tests                  |      actions
;-------------------------------------------------------------------------
SENTENCE
        +Q1   |   class AUXVERB ?         |  VERB:=word[ROOT]
              |                           |  TENSE:=word[TENSE]
              |                           |  TYPE:='QUESTION
;       ------------------------------------------------------------------
        Q2    |   none                    |  SUBJ<=NOUN-PHRASE
              |                           |  TYPE:='DECLARE
;-------------------------------------------------------------------------
Q1      Q3    |   none                    |  SUBJ<=NOUN-PHRASE
;-------------------------------------------------------------------------
Q2      +Q3   |   class VERB ?            |  VERB:=word[ROOT]
              |                           |  TENSE:=word[TENSE]
;-------------------------------------------------------------------------
Q3      +Q3   |   class VERB ?            |  put SUBJ on hold as NP
              |   is word PPRT ?          |  SUBJ:=('NP ('PRO 'someone))
              |   VERB='be                |  AGNTFLG:='TRUE
              |                           |  VERB:=word[ROOT]
;       ------------------------------------------------------------------
        +Q3   |   class VERB ?            |  TENSE:=TENSE+'PERFECT
              |   is word PPRT ?          |  VERB:=word[ROOT]
              |   VERB='have              |
;       ------------------------------------------------------------------
        Q4    |   is VERB TRANS ?         |  OBJ<=NOUN-PHRASE
;       ------------------------------------------------------------------
        Q4    |   holding NP ?            |  OBJ::=remove NP from hold
              |   is VERB TRANS ?         |
;       ------------------------------------------------------------------
        exit  |   is VERB ITRANS ?        |  <=('S ('TYPE TYPE)('SUBJ SUBJ)
              |                           |        ('VP ('TNS TENSE) ('V VERB)))
;-------------------------------------------------------------------------
Q4      +Q7   |   word='by                |  AGNTFLG:='FALSE
              |   AGNTFLG='TRUE           |
;       ------------------------------------------------------------------
        +Q5   |   word='to                |  none
              |   is VERB S-TRANS?        |
;       ------------------------------------------------------------------
        exit  |   none                    |  <=('S ('TYPE TYPE)('SUBJ SUBJ)
              |                           |        ('VP ('TNS TENSE)
              |                           |        ('V VERB) ('OBJ OBJ)))
;-------------------------------------------------------------------------
Q5      Q6    |   none                    |  SUBJ|::=OBJ
              |                           |  TENSE|::=TENSE
              |                           |  TEMP:='DECLARE
              |                           |  TYPE|::=TEMP
              |                           |  OBJ<=VERB-PHRASE
;-------------------------------------------------------------------------
Q6      +Q7   |   word='by                |  AGNTFLG:='FALSE
              |   AGNTFLG='TRUE           |
;       ------------------------------------------------------------------
        exit  |   none                    |  <=('S ('TYPE TYPE)('SUBJ SUBJ)
              |                           |        ('VP ('TNS TENSE) ('V VERB)
              |                           |              ('OBJ OBJ)))
;-------------------------------------------------------------------------
Q7      Q6    |   none                    |  SUBJ<=NOUN-PHRASE
;-------------------------------------------------------------------------

VERB-PHRASE
        +Q3   |   class VERB ?            |  VERB:=word[ROOT]
              |   is word UNTENSED?       |
;-------------------------------------------------------------------------

NOUN-PHRASE
        +NP1  |   class DET ?             |  DET:=word[ROOT]
;       ------------------------------------------------------------------
        +NP3  |   class NPR ?             |  NPR:=word
;-------------------------------------------------------------------------
NP1     +NP1  |   class ADJ ?             |  ADJS:=#ADJS+word[ROOT]
;       ------------------------------------------------------------------
        +NP2  |   class NOUN ?            |  NOUN:=word[ROOT]
;-------------------------------------------------------------------------
NP2     exit  |   none                    |  <=('NP ('DET DET)('ADJ #ADJS)
              |                           |        ('NOUN NOUN))
;-------------------------------------------------------------------------
NP3     exit  |   none                    |  <=('NP ('NPR NPR))
;-------------------------------------------------------------------------
```

Tactical Display Domain Example

The second example is part of a tactical display description abstracted from Sundfor (Sundfor, 1983a). The example has been extensively edited in the interest of space and is not a complete statement in the domain. In fact only a few statements in each of the major sections remain to give the flavor of the language.

```
/* Targetdisplay by Sigmund Sundfor                                          */
/*                                                                           */
/* Definition of a tactical display subsystem, called "Targetdisplay"        */
/* using the domain language for tactical displays on ship borne gun         */
/* control system.                                                           */
/*                                                                           */

Tactical display Targetdisplay :
   Configuration:
      relation own_ship [key id] of
         id,
         north,east,
         X_position,Y_position,Z_position,
         X_velocity,Y_velocity,Z_velocity,
         roll,pitch,heading,
         time_of_update;
   World model:
      relation classification [key id] of id, class;
   Access to world model:
      own_ship access is
        {temp4 := own.name join own_geographic_pos.name;}
         tctdisplay own_ship(name,north,east,X_position,Y_position,Z_position,
         X_velocity,Y_velocity,Z_velocity,roll,pitch,heading,time_of_update)
         := {temp4(name,north,east, X,Y,Z, XV,YV,ZV, R,P,C, clock);}
   Commands and parameters:
      on-off commands are
         [pb_display_only_hostile,
         pb_delete_id,
         pb_colour_hostile_red,
         pb_true_motion];
   Graphic representation:
      target graphic:
        vector length = 180 seconds ;
        length limit =  50 knots ;
        time between history points = 180 seconds;
        number of history points = 6;
        alpha-numeric is [digit(1..4) = target(number);];
        target symbols:
          [friend,submarine  = symbol((arc, -5,0, 0,-5, 5,0));
           friend,surface    = symbol((circle, 5));
           friend,air        = symbol((arc, -5,0, 0,5, 5,0));
           hostile,submarine = symbol((vector, -5,0, 0,-5, 5,0));
           hostile,surface   = symbol((vector, -5,0, 0,-5, 5,0, -5,0));
           hostile,air       = symbol((vector, -5,0, 0,5, 5,0));
           ];
      own_ship graphic:
         velocity vector type = point_to_edge_vector;
         speed marker / speed vector length = 180 seconds;
         time between history points = 30 seconds;
         number of history points = 6;
         alpha numeric speed and course display = true;
         screen position of speed and course display = (460,900);

/* The tasks are asynchronous and policy scheduled by priority *
   The tasks are:

      when command is pb_display_only_hostile
      then
         display all target where target_category is hostile
         every 1sec, priority 2;
      otherwise
         display all target every 1sec, priority 2;

      when command is pb_delete_id
      then
         show all target graphic except alpha-numeric;
      otherwise
         show all target graphic;

      when command is pb_colour_hostile_red
      then
         colour all target where category is hostile colour(1);
         colour all target where category is not hostile colour(2);
      otherwise
         colour all target colour(2);

      display all cursor every 100msec, priority 1;
      colour all cursor colour(3);

      display all own_ship every 1sec, priority 2;
      colour all own_ship colour(2);
```