Draco Domain Analysis for a Real Time Application: Discussion of the Results

by
Sigmund Sundfor
June 1983

Reuse Project RTP 016

Department of Information and Computer Science

University of California, Irvine

Irvine, CA 92717

This research was supported by NSF grant MCS-81-03718 with the cooperation of the Air Force Office of Scientific Research, and grants from A/S Kongsberg Vapenfabrikk and Royal Norwegian Council for Scientific and Industrial Research.

Table of Contents

1. Acknowledgments	1
2. Introduction	2
3. Evaluation of the Analysis and the Results	4
3.1 Using Draco	5
3.1.1 The Analysis Process	5
3.1.2 Unsolved Problems	7
3.1.3 The Draco Toolset	11
3.2 The Analysis Done	13
3.3 Applicability to Real-Time, Embedded Systems in General	17
4. A Procedure for Doing Domain Analysis	22
5. Summary	28
I. A short introduction to Draco	30
I.1 The Draco View of Software Production	30

List of Figures

Figure 3-1: Large systems has to be specified in terms of several domains 7
Figure 3-2: A suggestion for representing large systems as a set of specifications written in different domain languages. 9
Figure 3-3: A tactical display picture 13
Figure 3-4: Part of an example witten in the domain language. It is taken from the graphic representation and task definitions. 15

1. Acknowledgments

I would like to thank everybody who has helped me with this work. In particular, I would like to thank Peter Freeman who invited me to come to Irvine and suggested this work. He has guided and encouraged me along the way and reviewed the work as it has progressed. Jim Neighbors whose thesis is the basis for this work, has patiently explained the principles of Draco, helped sort out my problems with the mechanism and given me a lot of useful suggestions. Ira Baxter has helped me evaluate several of the ideas. Tom Marlin and Tore Aavatsmark helped review part of the SADT models. Martin Katz has helped me out of the numerous "fights" I have had with the computer system and the tools.

2. Introduction

Draco was developed by Neighbors [Neighbors 80] to demonstrate a way of capturing and reusing analysis, design and program components. The intention of this study has been to gain some experience in how well this approach works for real time embedded software [Freeman 82]. This is one out of two reports on the work done. The other report presents the domain analysis itself [Sundfor 83]This report discusses some of the results.

The motivation for doing this study is that software development has become a major cost for real-time, embedded computer systems. Reusing software workproducts may be a way of drasticaly increasing productivity. This is what Draco sets out to provide. Furthermore there is a shortage of people to do this software development. Those that are competent to do it, have to a large extent more knowledge of the application than of computer science. This does make the Draco approach interesting because it sets out to provide a tool (the domain language) that allows the system builder to define the system in the terms used in the application.

Originally the intention was to analyze real-time, embedded systems in general. This was found to be too wide an area to be treated as a domain. Instead, the application "ship borne gun control systems" was chosen. This is an area the author has some experience in from industry. The result of the initial analysis of this, was that also this application would be difficult to represent as a domain. It is reasoned in the other report that such systems would probably have to be represented as a collection of domains.

The domain that was finally chosen for the Draco domain analysis was that of the tactical plot in ship borne gun control systems. The domain analysis basically should result in a description of the objects and operators in the domain. This information is then captured and implemented in the form of a domain language. The analysis done includes the definition of the domain language syntax in form of the Draco parser definition, and the prettyprinter for printing out the internal form. To actually implement working systems, transformations and refinements for the domain will also have to be defined. This is not done in the present study.

This report discusses how well Draco worked for this application. It also considers to what extent Draco may be applicable to other domains in the real-time, embedded systems area. Some of the problems experienced in using the approach are presented. In addition, some possible problems that are foreseen based on the authors experience in developing such systems are mentioned.

The report includes a suggestion for how domain analysis can be performed based on this work. It presents the different stages of the analysis and discusses the activity that has to be performed.

3. Evaluation of the Analysis and the Results

Before the results are discussed, a few comments to this study may be useful: The purpose of this study has not been to demonstrate the effectiveness of Draco. Demonstrating that one actually can make systems and not just small examples was done as part of the original thesis work by Dr. Neighbors, and is also part of the research he is doing now. Dr. Neighbors work demonstrates the power of the approach and that it actually works. For this study, one could say: "Given that Draco provides a powerful approach and set of tools for developing systems, can it also be applied to real-time, embedded systems?" This study has aimed at giving some insight into this and particularily to find what special problems there may be for domains in this field when using Draco. It does therefore not discuss whether or why an approach like Draco can be a very efficient way of increasing productivity.

The intention has also been to gain more experience in doing Draco domain analysis. In particular, this study has given experience on how easy Draco is to use for somebody with application background (the author has several years of experience in development of real-time systems, but previously no formal computer science training).

3.1 Using Draco

3.1.1 The Analysis Process

Draco is really a set of principles. In addition it provides a set of tools for implementing these. These two aspects should therefore really be evaluated separately. The tools do necessarily bear some evidence of being the result of a research project rather than a fully industrialized project. The criticism of the user friendliness of the tools is not a reflection on the methodology principles.

The starting point of the Draco approach is the domain analysis. This was the work done in this study. This forces the user to do a proper analysis and record the results in a formal manner. It does in effect enforce the principle that most people in software engineering agree upon today, namely that more time spent "up front" in the development process saves money for the project as a whole. The domain analysis goes further than that in that it requires the analysis of a complete domain, not just a particular instance of that domain. The additional investment required by this is only justified if one will be building several systems in that domain, i.e. if the results can be reused.

The domain analysis did take quite a bit of time. One of the initial problems was appreciating what the domain analysis actually entails and what a domain is.(The meaning of the word "domain" was clear, but not the meaning in this context.) In retrospect, the domain analysis is not very different from an ordinary analysis except the fact that one is trying to analyze the whole domain. When the work started, this was not so clear.

Another term did also cause some initial confusion. The prettyprinter was first understood as being part of the end product made using Draco. Instead it is part of the tools like the parser that help a system builder make systems using Draco.

When one does not fully understand a concept and how it works, it is difficult to bound the problems. The knowledge gained on Draco during talks by and with Neighbors and Freeman helped a lot in this respect. This problem will probably be alleviated for other domain analysts as more examples and documentation are developed.

Quite a bit of time was also spent on cut and try. There was no recommended procedure for doing the domain analysis. It can be argued that one just should do it like any other analysis in software development. But, first of all there is not just one way of doing analysis. Secondly, the main workproduct from the analysis, the domain language, influences the form the analysis takes. Later on in this report, a procedure is suggested based on this experience.

Once the above problems were handled, the analysis went well. The quality of the workproduct will have to be evaluated by others who know the domain, but it was possible to capture much of the domain knowledge for the tactical plot in a language.

3.1.2 Unsolved Problems

The original model for building systems using Draco was something like: A system builder will express the system specification in a domain language. This specification is then transformed (simplified) and refined into other underlying domains. There can be a whole network of domains. The domain language that the system builder uses initially does in effect serve as a formal specification language. The language should as far as possible use the notations of the application.

This study indicates that this type of system cannot conveniently be expressed as one domain. A ship borne gun control system is what Lehman and Belady [Belady & Lehman 79] describes as "a large system". It essentially lies outside the grasp of a single individual. It will require an organized group of people to design and implement it. This is not so much a reflection on the size of the final program. Draco is indeed a tool that can make us handle that. However, it can be characterized as large because it encompasses several application areas. Since the domain language in effect is a specification language, it would have to be a very large language to capture all the different application knowledge necessary to specify the system. The conclusion was therefore that this type of system will have to be expressed in terms of several domains at the top level. This is discussed further in the other report on this study.

If the system is specified in terms of several domains, the problem of how these various parts shall communicate arises. It is conceivable that some parts that are typically "one-of-a-kind" applications, will be most efficient to program in the implementation language. This may make the problem of communication between the different parts even more difficult. The domain analysis done suggests a type of interface for that domain. But, the suggestion is not very elegant since it takes a lot of space and work to specify the interface and some low level constructs. The components (refinements) to implement it, are not developed and tried out. The solution

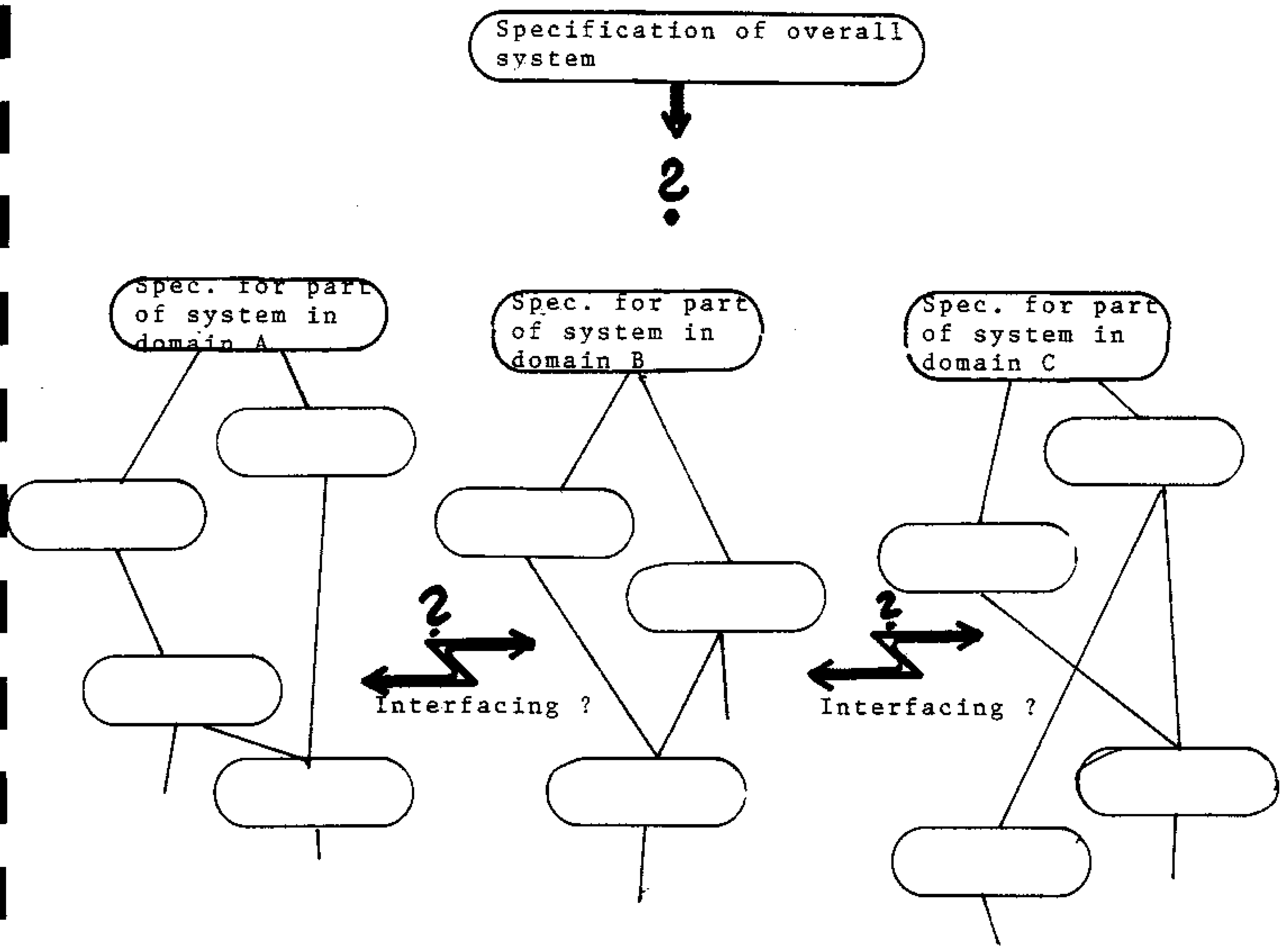

Specification of overall system
?
Spec. for part of system in domain A
Spec. for part of system in domain B
Spec. for part of system in domain C
?
Interfacing ?
?
Interfacing ?

Figure 3-1: Large systems has to be specified in terms of several domains

is not a general solution either. The problem is therefore basically unsolved. In addition, there is the related problem of how to represent the total system structure.

One may argue that "large systems" (as used above) need not be built using several domains at the top level. The decisions during refinements may in many respects be viewed as adding to the original specification written in the domain language. One could therefore conceivably start off with a specification of the systems in a domain that in very broad lines described the system. The specification for the different domain areas would then be added on as choices made during the refinement process.

However, guiding the refinements Draco does, is different from writing a specification in a domain language. The refinement mechanism as it is today, relies on a close interaction with the machine doing the refinements and some understanding of the Draco mechanism. Decisions have to be made there and then and committed. There is not any possibility for backtracking, except for starting all over again. The creation of a specification is a complex process. It is probably done better when done in a manner allowing the builder to reiterate and change as the specification develops. The refinement mechanism is therefore not a suitable means for giving the system specification.

If instead of interactivly guiding the refinement process, the system builder could write the guidelines using a domain language, this situation would change. In a way this could be a further development of the strategy mechanism. If this was made possible, a "large system" may be expressed in a hierarchy of specifications using Draco domain languages.

Testing is another problem that probably will show up using Draco in the present form for real-time, embedded systems. These systems include special purpose equipment and interfaces. This does mean that there will be a need for testing new components fairly close to machine level. Even if it should be possible to test components reasonably well as freestanding parts, they may have undetected faults, or the hardware may not function quite according to specifications. The need for testing at this level may (will) therefore arise after systems are built. This appears very difficult to do the way Draco works today.

Draco will probably increase productivity dramatically. However, refining the original specification is still quite a bit of work. Therefore, if a minor change is done in the specification, it would be desirable to be able to

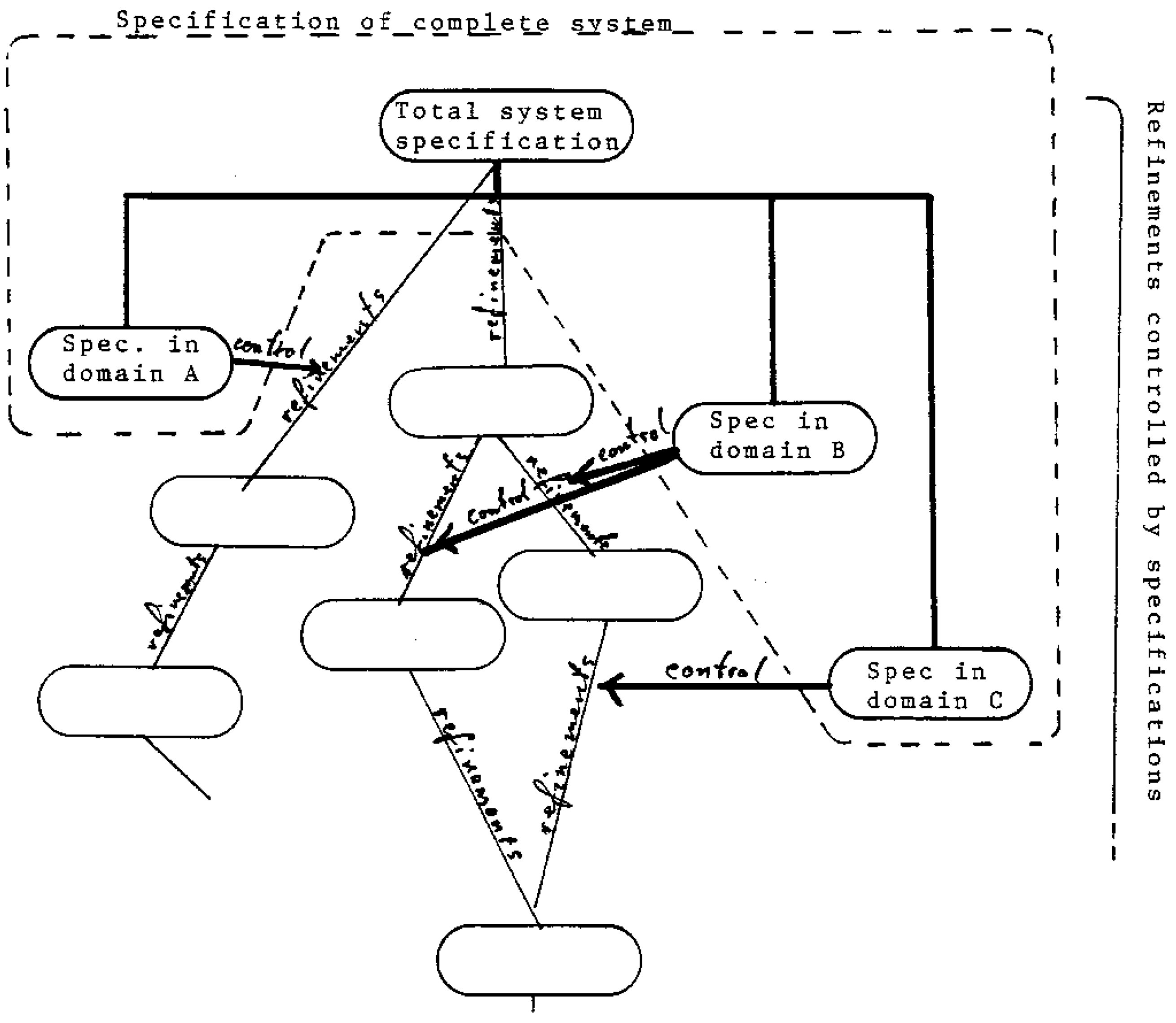


Figure 3-2: A suggestion for representing large systems as a set of specifications written in different domain languages.

replay the refinements without any interaction for all those parts unaffected by the change. This is not possible today. This is a hard problem to solve. Draco would have to determine the effect of the change to see what could be replayed. But, from a Draco user's point of view, it is certainly desirable.

3.1.3 The Draco Toolset

Draco is basically a set of principles and a mechanism for implementing these. In addition, a set of tools is provided to aid the user of Draco. These tools are themselves made using Draco. The tools used during the study have been those for defining the domain language syntax, namely the parser and prettyprinter builders.

The power of these tools is very good. It has been very simple for a person (the author) without prior experience in building parsers, to convert the syntax definition into a working parser. The same applies for the prettyprinter.

It was originally thought that these tools could be used without an understanding of some compilation techniques. The experience is that this does not hold true. As Draco is now, it appears necessary to understand at least what a parse tree is and how the transformation and refinement mechanisms work; at least in principle. A domain specialist without this knowledge would have difficulties creating and using a Draco domain without this understanding. The documentation and perhaps the user interface could be improved to simplify this.

The domain analyst and domain builder does not necessarily have to be the same person. However, the domain analyst will probably need this understanding of Draco to be able to communicate with the domain builder. It is also questionable how efficient the splitting of the two jobs between two persons may be. Draco does not provide any formalism for communicating between the analyst and builder. What is provided, is a mechanism for the analyst to record in a formal manner the results of the analysis. If it is difficult to find people who can cover both the application and the sufficient understanding of Draco, then it is probably better to provide more tools in Draco and/or have a Draco consultant to aid the analyst build the domain.

Given the necessary insight, the tools are simple to use. There is of course some cryptic parts which probably can be accounted for by the fact that it is a research project. The error messages especially, could do with some refinements.

There are some additional tools and features that may be desired. Some suggestions are listed below:

- Some simpler, hopefully semi-automatic means for generating error recovery blocks in the parser. The parser made for this language has no error recovery because it would have taken a large amount of work to implement.

- A syntax directed or assisted editor (or generator of such) using the parser and prettyprinter definition for the syntax definition.

- A better means for checking consistencies of specifications written in a domain language. At the moment, some simple checking can be done by the parser using the "USE" and "DEF" functions. However, a need is felt for being able to specify stronger constraints on the input. One example is the "strong typing" defined in some programming languages. Other types of static analysis would also be advantageous.

- An improvement in the user interface during transformations and refinements. Instead of looking at a printout of the internal form tree, the user should see the instance she/he is at in the tree and the surrounding as a piece of code in the language for the domain this instance is in. It would be nice if it looked to the user as she/he was moving round in a piece of code rather than an internal form tree. (For most application specialists, a tree is something that grows out in the garden or forest.)

3.2 The Analysis Done

The analysis done resulted in the definition of a domain language for the part of ship-borne gun control systems known as the tactical plot. It resulted in a parser and prettyprinter definition. The transformations and refinements are not defined, but the graphic components are described.

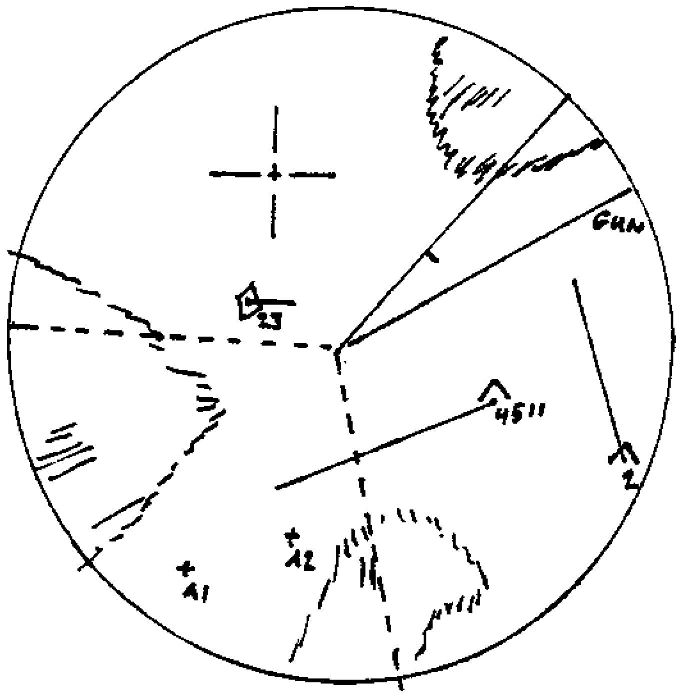


Figure 3-3: A tactical display picture

The work started off with modelling the domain using SADT [Ross 77]. Thereafter, quite a bit of time was spent on deciding upon the form of the language. It was difficult to decide upon what kind of abstraction it should provide. Most abstraction of the base machine limits what the user can express as discussed by Siewiorek and Parnas [Parnas&Siewiorek 72]. On the other hand, one does not want to burden the user with unnecessary decisions on how functions should be implemented.

The two types of languages considered were a procedural form with powerful primitives relevant to the application and a non-procedural specification language. The procedural type would be somewhat similar to conventional programming languages in structure, but embodying functions like transformation, clipping, frequency of update, and knowing about basic data types like real world coordinates, and velocity. The system builder would have to specify how the data should be processed: What objects there are and how they should be represented. This would clearly be the most powerful language in terms of the range of systems that could be represented and range of implementations.

On the other hand, it was felt that the information on what kind of objects are shown on a tactical display and how they are represented, is part of the domain knowledge. Furthermore, it should not be the concern of the user of the domain language how the data are processed to generate such tactical plots. Therefore it was decided to define a non procedural language along these lines. The system builder can define which objects are to be included from a set of predefined objects (targets, guns etc.). The basic shapes of the graphical representation is fixed, but the system builder specifies some parameters and conditions for displaying and colouring. How the processing is done will be determined by the components (refinements) for the domain and need not and cannot be controlled by the specification written in the language. The language does provide some limited facilities for defining new objects and their representation.

The approach chosen does necessarily limit the range of systems that can be represented. It does in many ways resemble an application program generator input form. On the other hand, it was difficult finding a set of useful primitives that could be incorporated in a procedural language such that it would both be easy to build tactical plot subsystems and giving the flexibility to define any kind of objects and graphical representation for them. Flexibility will very often mean increased burden on the user of the language.

The aim has been to use the terminology of the application. It is hoped that the part dealing with the specifications giving parameters for the graphic representation and the part defining the conditions for displaying and the colouring, should be readable for the customer. (These are the "graphic_rule", the "task_rule" and part of the "configuration_rule" in the syntax definition.) These parts of the specification written in the domain language, could conceivably serve as formal specification for the tactical plot in the contract with the customer. This is of course a personal opinion.

However, the language has also to deal with the previously mentioned problem of the communication with parts written in other domains. This took quite a bit of work. It takes up quite a bit of the syntax. The first four sections (configuration_rule, world_model_rule access_rule and command_rule) deal primarily with this. It is not felt that this is solved satisfactorily.

The form chosen for the language has led to a very large syntax definition. All objects of the domain have at least one syntax rule. The volume has further increased by the attempt to make the specifications written in this language readable for the domain specialists. This means that the syntax hardly can be expected to be learned by heart by any system builder. On the other hand, the syntax has not a lot of recursive constructs leaving a wide variety of combinations open. The syntax definition can therefore best be used as a guideline the user of the language keeps in front of her/him while writing the specification. It would be even better if the syntax was provided as something like a template or a syntax directed editor. (A syntax assisted editor that allows the user to have illegal constructs hanging around while editing, is probably more user friendly than a strict syntax directed editor.)

Most likely there are some errors and inadequacies in the language definition. Some of these will probably be evident when the transformations and refinements are implemented. Some weaknesses are known. The syntax definition does for example, require an exact number of spaces between the words in the strings between quotes in the syntax definition.

The work preceeding the language definition itself was reviewed by a couple of persons with knowledge in the domain. The actual syntax and the examples

```
      .
      .
      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));
      .
      .
      cursor graphic:

      cursor(1):
         cursor symbol = symbol((vector, 2,0, 10,0)(vector, 0,2, 0,10)
                         (vector, -2,0, -10,0)(vector, 0,-2, 0,-10));

         true motion = pb_true_motion;
         center cursor = center_cursor;
         own ship to cursor = false;
         cursor to own ship = false;
         cursor movement X = cursor_inc(X);
         cursor movement Y = cursor_inc(Y);
         .
         .
   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_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);
      ;
      .
      .
```

Figure 3-4: Part of an example witten in the domain language. It is taken from the graphic representation and task definitions.

written have not been reviewed by any other specialist in this domain. This should be done if the language should actually be implemented.

3.3 Applicability to Real-Time, Embedded Systems in General

The study done has been applying Draco to a particular domain within the field of real-time systems. It is fairly simple to see that it can readily be extended or changed to cover the domains of tactical plots for other type of systems than ship-borne gun control systems. It covers very many of the same aspects as found in the tactical plots of simple CCIS systems used aboard ships. Other CCIS systems have different and more extensive functions, but they are not that different that they could not be represented by an elaboration of the present domain or a new one along these same lines using Draco.

Tactical plots represent one application, but real-time, embedded systems cover much more. Part of the purpose of this study was to try out Draco on a domain from this field. The results indicate that it can be used. It is interesting to consider whether it has wider applicability in the real-time, embedded field than this one domain demonstrated here. To do this, it may be useful to consider some of the characteristics of such systems.

Real-time, embedded systems are often thought of as a very special field of computer applications. "They are difficult with a lot of special purpose processing, rigid constraints and a lot of low level bit manipulation. They are best coded in some low level programming language, preferably assembler, where one has full control of what goes on without the confusion introduced by some higher level tools." This is the impression one may get from observing what goes on in the field. The contention is that this is not so. There are certainly some aspects of the field that are especially difficult and requires special attention. But, a large portion is not significantly different to other computer applications.

The following are some characteristics that it is believed hold for many of the real-time embedded applications and the software development for these:

- Real time response.
- Embedded.

- Large systems in the terms of Belady and Lehman.
- Implements advanced technology.
- Implemented by application specialists (rather than computer scientists)

There may be more, but let us consider these.

Being real time means that a system has to process information fast enough to react to and/or control events in the outside world in a time frame that permits the real world events to be affected by the system. This puts constraints on how long time processing can take and may also require synchronization with external events. In general, there is only a small part of the system that has rigid time constraints. Even these may not always be that rigid. It is often sufficient that these can read a real time clock and modify the calculations accordingly when they control and/or monitor physical processes where time is a factor. There are also many times when there is very little freedom in when the results of computation must be ready. However, this is not really all that difficult a problem. It requires simple synchronization techniques and also that one does not have operating systems that suddenly shuts the world off for some seconds while it goes off on its own for some garbage collection or similar internal tasks. The real problem with the real time aspect is often too many tasks for too little computer capacity.

Embedded means that the computer and software is part of a larger system whose prime purpose is not the processing itself. It generally leads to a lot of special purpose equipment and interfaces to them that has to be handled. The variety is probably much larger than need be. A lot could be gained by some careful analysis of this field. It is for example difficult to see that there is any good reason for representing digitized analogue signals (with the same resolution) in up to four different ways except that this is what IC package producers provides. There is also an enormous variety in "standard" protocols for communicating with equipment and interfaces plus all the equipment that does not follow any other protocol than its own. We will probably have to live with this situation for some time to come. Some special purpose equipment will probably always have to be handled.

This will require that we have the ability to deal with machine near details. It is important to note though, that this, like the real time aspects, only affects a relatively small part of the software. It can probably be even less than it is today by some careful hardware design.

"Large systems" relates, as previously mentioned, to the variety within the system. A system is large if it essentially lies outside the grasp of a single individual and requires an organized group of people to design, implement and maintain it [Belady & Lehman 79]. This is not the case for all real-time, embedded systems, but it applies to a large portion. This does lead to added complexities as is noted by Belady and Lehman, that is not well handled by today's methods. This is a strong argument for applying methods that can provide more rigorous control of the software structure and higher level definitions of what the system should do.

The restricted resources of the embedded computers both in time (processing power) and memory, often leads to the choice of low level languages or assemblers to allow optimizations. It is questionable whether this is the correct choice. It may allow optimizations at the detailed level, but the gain can easily be lost several times over in inefficiencies at the macroscopic level. It is difficult to express higher level constructs and controls in assembler or low level languages.

Many of the systems in this field are implementations of advanced technology. This means that they will have a kind of experimental flavour. There will be a lot of cut and try with changes to the system. This may mean that some domain knowledge will be difficult to capture in a Draco domain language because the domain itself is not fully understood. On the other hand, it increases the usefulness of having powerful functions for building systems and redoing them when needed, as Draco provides.

The last aspect noted, is that the people writing the software for these types of systems very often are application experts rather than computer scientists. This is only an observation based on the impression of the author and an informal survey that was done in Norway. However, as far as this is correct, these are people who primarily regard the software as a tool for implementing their applications. The reluctance to move away from assembler amongst some, is probably part fear of change, part lack of understanding of the concepts presented to them by the computer scientists in the form of new, even more complex language constructs that appear to bear little relevance to their applications. Draco has the advantage here that it aims at providing domain languages that are directly relevant to the applications.

This section has tried to argue that real-time, embedded systems for a large part are not all that different from other computer applications. Parts of the systems are special and may require special treatment. Most of the domains, however, are not all that different from other computer applications. It will therefore be reasonable to expect that the results from this and other studies on using Draco will apply to other real-time domains as well. Furthermore, since many of these must be denoted as large systems, it will be advantageous to introduce tools that makes the complexity more manageable. The people building the systems should also be aided by having a tool using the terms of the domains they are experts in rather than traditional programming languages or assemblers.

All real-time embedded systems do not come under the descriptions above. A notable exception is those that control very critical processes and require ultra high reliability. These are given quite a bit of attention by the research community today and do probably require other types of tools than what Draco provides for the development.

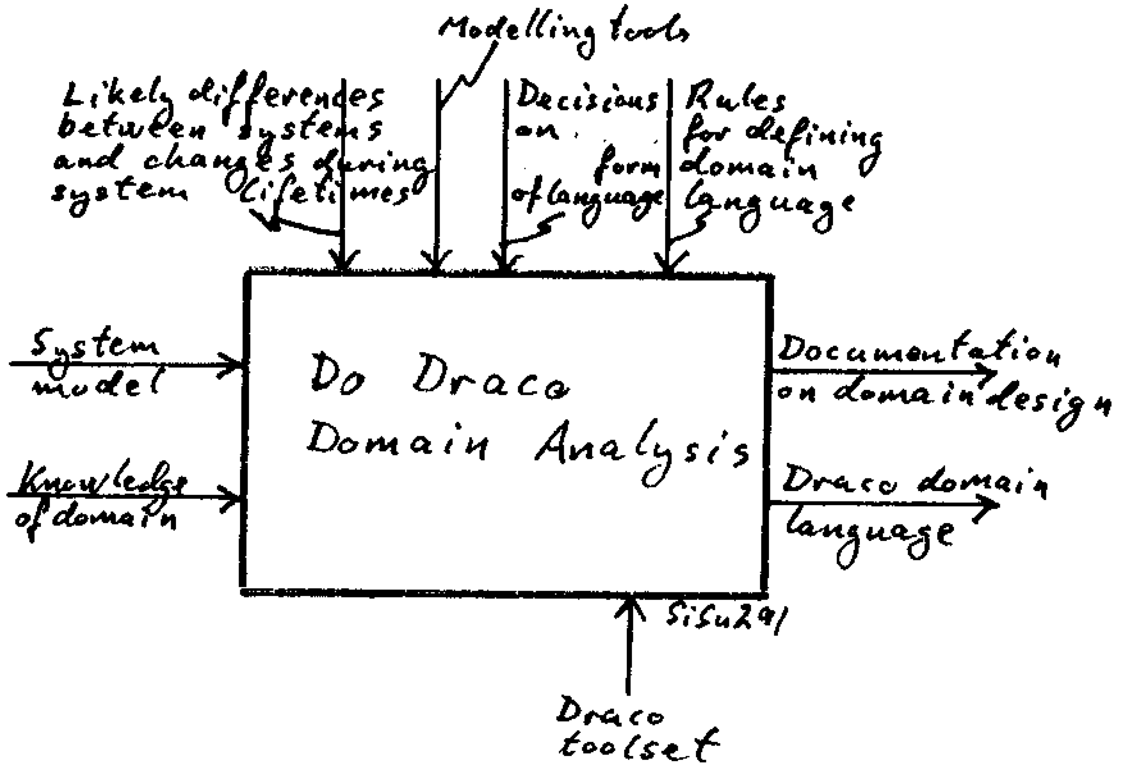
The larger portion of the real-time systems though, are developed today with low level tools and ad hoc methodology. There is a large room for improvement as has been noted by both the DoD's Ada and STARS programs [DoD

83]. It is important that the methodologies and tools applied cover the whole life cycle [Freeman and Wasserman 83]. Draco does set out to cover the life cycle (although not all aspects). There are some problems left, but there is good reason to believe that approaches like Draco can drasticly improve the development and evolution of such systems.

4. A Procedure for Doing Domain Analysis

The domain analysis done involved a lot of cut and try. This can probably not be avoided. On the other hand, the experience gained in doing this, suggests some order in which things can be done. This is illustrated in the following SADT model. The steps are further described in the text following the diagrams. This is not necessarily the only or even the best procedure, but it appears to work based on the experience gained from this study.

USED AT:	AUTHOR: Sigmund Sundfør PROJECT: Draco Domain Analysis NOTES: 1 2 3 4 5 6 7 8 9 10	DATE: June 6-83 REV: June 24-83	[✓ WORKING] [✓ DRAFT] [✓ RECOMMENDED] [✓ PUBLICATION]	READER DATE	CONTEXT: None


Modelling tools
Likely differences between systems and changes during system lifetimes
Decisions on form of language
Rules for defining domain language
System model
Knowledge of domain
Do Draco Domain Analysis
Documentation on domain design
Draco domain language
SiSu291
Draco toolset

Viewpoint: The person doing the domain analysis

Purpose: To illustrate a procedure for doing domain analysis. It illustrates the situation where a system will be defined in terms of several domains

Type of model: Illustration of the activities the domain analyst must perform

NODE: Procedure A-0	TITLE: Do Draco Domain Analysis	NUMBER: SiSu 290

SADT™ DIAGRAM FORM ST098 9/75

USED AT:

AUTHOR: Sigmund Sundfør
PROJECT: Draco Domain Analysis

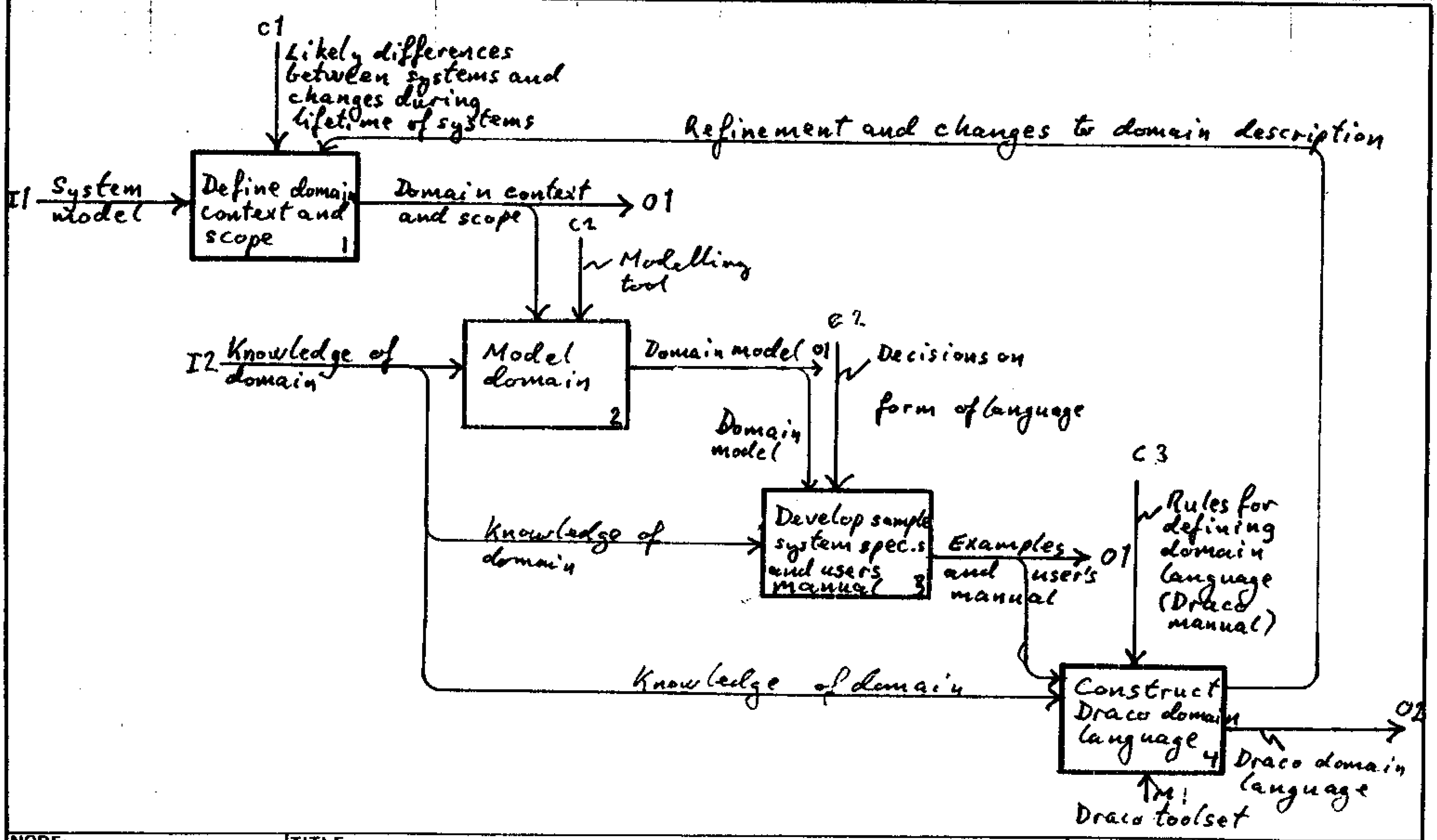
DATE: June 6 - 83
REV: June 24 - 83

NOTES: 1 2 3 4 5 6 7 8 9 10

[✓ WORKING]
[✓ DRAFT]
[✓ RECOMMENDED]
[✓ PUBLICATION]

READER DATE

CONTEXT: Top


C1
Likely differences between systems and changes during lifetime of systems
Refinement and changes to domain description
I1 System model
Define domain context and scope 1
Domain context and scope
O1
C2
Modelling tool
I2 Knowledge of domain
Model domain 2
Domain model O1
C2
Decisions on form of language
Domain model
Knowledge of domain
Develop sample system spec.s and users manual 3
Examples and user's manual
O1
C3
Rules for defining domain language (Draco manual)
Knowledge of domain
Construct Draco domain language 4
O2
Draco domain language
M1
Draco toolset

NODE: Procedure A0

TITLE: Do Draco Domain Analysis

NUMBER: SiSu 291

The underlying idea of the procedure is that the domain analysis requires that one develop a thorough understanding of the domain. The starting point is a domain expert doing the analysis. Since he/she is already an expert, it should not be necessary to develop any further understanding. It is not quite that simple. Even a domain expert will have good use of tools for organizing and recording the knowledge. In addition to aiding the expert itself, such tools will make it possible to communicate the ideas to other people with knowledge of the domain and get feedback from these.

The domain definition using Draco, is in itself a tool for recording the domain knowledge. The purpose is to do this so that the information can be reused. For some domains the knowledge is probably best recorded directly as a Draco domain definition. For this domain studied here, and probably many others, some steps before the domain definition itself is advantageous.

The first, fairly obvious step, is to determine what the domain is. One needs to determine what it should cover and the context to other domains. The tactical plot domain studied here covers only part of a larger system. In this case, it is very important to determine what is within the domain and the relation to the rest of the system. Other domains may cover complete systems like reservation and information systems for travel agencies. But even in this case it is important to determine the range of possible systems and features one will cover. This is an iterative process, and the definition of what is covered by the domain, will be refined as the analysis proceeds. (The end product is the objects and operations in the domain and the relation between them, as expressed by the domain definition.)

In this study, a model of the domain of the whole system was first developed using SADT. It was a good tool for this purpose and it allowed the work to be reviewed by several other people and thereby giving very useful new input and ideas. Other applications may have other modelling techniques that are more known or especially suited to the field. Whatever the technique is, modelling is a useful way of organizing and improving the understanding of applications.

In this case, the type of systems considered was found necessary to represent as several domains even at the top level. In the other report on the analysis [Sundfor 83], it is argued that this may be best done according to the principles advocated by Parnas [Parnas 79].

Once the extent and context of the domain has been reasonably well determined, it is time too take a closer look at the domain itself. Here again it was advantageous to develop a model both for organizing one´s own understanding and for communicating the ideas to others for them to review and giving their knowledge.

The next step suggested, is deciding upon the form of the language itself. This was the stage that gave the most problem in this study. The first attempt was writing down the syntax straight away. This did not work out. First of all, with little language theory and parser builder background, the attempt soon ended up in a lot of petty language problems. Secondly, both the form and the type of abstractions to be used were only vague ideas. The next attempt was sitting back and trying to envisage the situation of the user of the language: "If I was going to build a system using this domain language, how would I like it to be?". There is nothing revolutionary in this approach, it is called writing the users manual before building the system, as was pointed out by Peter Freeman [Freeman 83]. This is indeed what was done. But, even before that, a few examples were worked out writing system specifications the way the imagined user of the domain language would like to do it. The user´s manual was written based on these examples and the ideas behind them.

The form and abstractions of the domain language are not simple to decide upon. In most cases there will be compromises. Very domain-specific constructs will may make it simple to define systems. On the other hand, they probably limit the range of systems that can be constructed. The domain analysis is a very expensive task, so there are also limits to how much time one can spend at perfecting the domain language. A useful language is usually better than a "nearly finished," perfect language.

The final step is the definition of the domain language itself using Draco. The syntax definition of the language is recorded in the parser definition. In addition, a prettyprinter is needed to print out the internal form. The source to source refinements provide simplifications of constructs in the domain while the refinement actually implements the components of the domain by refining them into underlying domains. These steps are described in the Draco documentation.

5. Summary

The work done in this study has been:

- The domain focused upon, was narrowed down from real-time systems in general to ship borne gun control systems, and from there to the tactical plot subsystem of the gun control.

- The gun control system domain and the tactical plot domain were modelled using SADT. The modelling was done both to organize and refine the knowledge of the domains.

- A domain language was constructed through several iterations of trying by hand different constructs and writing a user's manual.

- The language syntax has been defined in the Draco system by building a parser and prettyprinter for the language. The semantics are represented by the domain model and the prose description. The semantics have not yet been defined in Draco, i.e. the transformations and refinements are not built.

- The parser and prettyprinter have been tested out on an example that was developed during the design of the language.

This domain analysis covers a domain from the real-time, embedded application. There are some problems not yet dealt with by Draco that has to be solved. Apart from these problems noted in the study, Draco appears feasible for this real-time domain. Furthermore, most of the real-time domains are not so special that they cannot use methods found useful for software developments in other domains. It is therefore reasonable to expect that as far as Draco can be demonstrated as being useful applied to other domains, it will be useful to a large part of the real-time, embedded domains as well.

The form chosen for the language of the domain, is non-procedural with a lot of domain specific constructs. It aims at being descriptive rather than prescriptive. It looks very little like traditional (Algol-like) programming languages. It is more meant to mirror the way specifications for tactical plots actually may be written in a contract with the customer. In addition, it deals also with the basically unsolved problem of interfacing to other parts of the system. This complicates the syntax. The syntax definition is very long and may therefore be hard to use. Some simple tools directing the user would help in such a case.

The domain analysis itself does involve a lot of work. A procedure is suggested for doing this based on the experience gained in this study.

It is worth noting that a domain analysis is not something that is peculiar to Draco. In my opinion, anybody building software systems would gain from analyzing the domains they are working in. What Draco provides is a means of recording the results of this process. In addition it provides a mechanization of the reuse of the results. Even without this mechanism, the recorded results of such an analysis can be used. It conveyes a lot of information on the domain and can even be considered used as a way of formally specifying systems in that domain.

Based on the experience gained in this study, the following work on Draco is recommended:

- Mechanisms for handling the interfacing between parts of large systems specified in different domain languages.
- Tools for testing code in systems built using Draco. This may in particular be necessary for testing special purpose interfaces in embedded systems.
- Refinement replay mechanism.
- More static analysis tools that can be incorporated in the languages.
- Syntax directed or assisted editor working from the parser and prettyprinter definitions.
- Automatic or semi-automatic generation of error recovery blocks in the parsers (by the parser builder tool).
- Improved error messages from the Draco mechanism and tools.
- Improved user interface during transformations and refinements.

I. A short introduction to Draco

It has been a common practice to name new computer languages after stars. Since the system described in this manual is a mechanism which manipulates special purpose languages it seems only fitting to name it after a structure of stars, a galaxy. Draco[1] is a dwarf elliptical galaxy in our local group of galaxies which is dominated by the large spiral galaxies Milky Way and Andromeda. Draco is a small nearby companion of the Milky Way ($1.2x10^5$ solar masses and 68 kiloparsecs from Earth). This small size and close distance to home is well suited to the current system which is a small prototype.

I.1 The Draco View of Software Production

The Draco system addresses itself to the routine production of many systems which are similar to each other. The theory behind its operation is described in detail in [Neighbors 80].

Three themes dominate the way Draco operates: the use of special-purpose high-level languages for the domains or problem areas in which many similar systems are needed; the use of software components to implement problems stated in these languages in a flexible and reliable way; and the use of source-to-source program transformations to tailor the components to their use in a specific context. The basic steps in the production of a specific system using a Draco supported domain-specific high-level language is as follows:

1. An analyst with experience in developing many systems in a certain problem domain decides that the domain is understood well enough to define a language suitable for comfortably and easily describing other systems in the problem domain. This person is called the Domain Analyst and the language described is called the Domain Language. The Domain Analyst describes the domain and its internal form with the parser generator part of the BUILD subsystem of Draco which is described in the Draco user′s manual.

2. Once the Domain Analyst has described the external and internal form of the domain then how program fragments in the domain should be printed so that users find them easy to look at and accurate in their meaning must be described. This is called prettyprinter

[1]Draco is Latin for dragon

generation and it is done by the Draco BUILD subsystem.

3. The Domain Analyst must provide simplifying relations among the objects and operations of the domain. These are used for simplification and optimization of programs in the domain. These simplifications are accepted in terms of source-to-source program transformations by the BUILD subsystem which forms them into a library of transformations.

4. Finally, the Domain Analyst must prepare a prose description of the meaning of the operations and objects in his domain.

5. This prose description is turned over to a Domain Designer who specifies components for the objects and operations in the domain which refine the objects and operations of one domain into other domains known to the Draco system. These components are formed into libraries by the Draco subsystem. A component is a set of refinements each capable of implementing a domain object or operation under certain stated conditions while making certain implementation assertions.

6. A new system which can be described in a Domain Language known to Draco can inherit some analysis, design, and coding from the Draco library. The statement of the system to be constructed is cast in a Domain Language. The Domain Language program is then turned into an internal form by the PARSE subsystem. This internal form is then given to a System Specialist.

7. The System Specialist interacts with the transformation and refinement subsystem of Draco. The basic operation in this phase is the selection of an appropriate set of software components to implement the operations and objects in the domain which are used in the problem statement. Then these components are specialized by program transformation to the problem at hand and then separately refined into another (or the same) domain and the cycle begins again. The refinement subsystem allows the definition of refinement tactics capable of removing the burden of answering low-level questions from the System Specialist.

8. The process the System Specialist uses to refine the problem is, of course, not strictly top down but the refinement subsystem keeps a record of the process which makes it look top down. When the program is in an executable form it is printed out by the System Specialist and either acceptable or the specification cycle begins again with the existing Domain Language program.

9. The refinement history of a program may be examined by a user of the EXAMINE subsystem which states what refinements were used in the production of this program. A higher-level description of all parts of the program to whatever level (up to the original Domain Language) always exists in the refinement history. It is hoped that these higher levels of abstraction of an existing program will be useful in understanding the program during the maintenance phase of its lifecycle.

The process described briefly above is dealt with in more detail in [Neighbors 80] which presents an SADT[2] model of the process.

---

[2]SADT is a registered trademark of SofTech Inc.

REFERENCES

[Belady & Lehman 79]
Belady, L.A. and Lehman, M.M.
The Characteristics of Large Systems.
In Wegner, P., editor, Research Directions in Software Technology, chapter 1, pages 106-131. The Massachusetts Institute Technology Press, Cambridge, Mass., 1979.

[DoD 83]
U.S. Department of Defence.
Software Technology for Adaptable, Reliable Systems (STARS) Program Strategy.
ACM SIGSOFT Software Engineering Notes , 8(2):56,108, April , 1983.

[Freeman and Wasserman 83]
Peter Freeman and Anthony I. Wasserman.
Ada Methodologies - Concepts and Requirements.
ACM SIGSOFT Software Engineering Notes , 8(1):33,50, January , 1983.

[Freeman 82]
Freeman, Peter and Neighbors, James Milne.
Reusable Software Engineering, a Proposal Submitted for Consideration by the National Science Foundation.

[Freeman 83]
Peter Freeman.
Private communications.

[Neighbors 80]
Neighbors, James Milne.
Software Construction Using Components.
PhD thesis, UCI, 1980.

[Parnas&Siewiorek 72]
Parnas, David L. and Siewiorek, D.P.
Use of the Concept of Transparency in the Design of Hierarchically Structured Systems.
Technical Report, Carnegie-Mellon Un.., July, 1972.

[Parnas 79]
Parnas, David L.
Designing Software for Ease of Extension and Contraction.
IEEE Transactions On Software Engineering , SE-5(2):128,137, March , 1979.

[Ross 77]
Ross, Douglas T.
Structured Analysis(SA): A Language for Communicating Ideas.
IEEE Transactions On Software Engineering :?, January , 1977.

[Sundfor 83]
Sigmund Sundfor.
Draco Domain Analysis for a Real Time Application; the Analysis.
Technical Report RTP 015, Department of Information and Computer Science, University Of California, Irvine , June, 1983.