Domain Analysis and Generative Implementation

James M. Neighbors
Bayfront Technologies, Inc.
1280 Bison B9-231
Newport Beach, CA 92660
neighbor@BayfrontTechnologies.com

1. The process

One of the most helpful and harmful concepts in early Software Engineering was the "waterfall" model of software development [1]. The model was successful in relating the general activities of software development. However, the model was a failure in providing a software management technique. Management misinterpreted the parallel steps of the model as sequential steps that could be used as project milestones. From the management view, system development moves strictly from analysis to design to implementation. From the developer view, system development exists in analysis, design, and implementation all at the same time, with the hope that towards the end of a project there is more implementation than analysis.

I believe there is a similar misunderstanding in how a particular system specified as a member of an analyzed domain evolves to an operational system. Is there a sequence of instantiated features to generic architecture to reusable assets through component glue to application generator? I suspect not – for reasons similar to the waterfall model. This is what happens when the output from one tool is the input to another tool. Instead, I believe there is a *progressive deepening* [2] by the system implementers of the system description, which is guided and restrained by the domain designers. Thus, subsystems deemed important by the system implementers are first refined (implemented), causing other subsystems to be constrained by the representation decisions made during the refinement. The architecture of the system evolves with these decisions [3]. Standard architectures are guides but not restrictions. Otherwise, what would one do with a system that composes two standard architectures?

2. What results from domain analysis?

The result of domain analysis is a domain that supports the description of similar member systems. In Draco a domain consists of the following definitions [4]:

1. **Parser** – Described in augmented BNF, explains how to convert a system description to an *internal form*. The language described by the parser is the *domain language*.
2. **Display** – Describes how to display internal forms as text and graphics for system implementers.
3. **Optimizations** – Describe the rules of exchange between statements in the domain language. Optimizations may be procedural or source-to-source.
4. **Components** – Specify the semantics of the domain by implementing constructs of the domain language in the domain language of other domains. Each component provides one or more *refinements* that restate the construct in different domains.
5. **Generators** – Describe programs that transform a statement in the domain language into another statement in the same domain.
6. **Analyzers** – Describe programs that gather information about the internal form for use by the other domain parts.
7. **Strategies and Tactics** – Describe plans for refinement based on available refinements and domain interconnections.

The domains are manipulated by Draco with guidance by system implementers. This basic scheme is over 18 years old. Why after so many years are we still talking about Draco? I believe there are three basic reasons.

First, Draco actually describes an individual notation, the domain language, for each domain. Consider that the sequence (A B) and alternation (A|B) operators of BNF (a domain language in itself) are equivalent to the AND/OR constructs of feature trees used by many DA methods. However with an augmented, recursive BNF the class of descriptions is infinite. Further, consider the utility of language – it restricts the allowed combinations. A simple component library says nothing about how the components may be combined. This effort, placed on each user of the library, represents a huge and unnecessary education effort. FORTRAN, Java, SQL, and OpenGL are successful as languages and protocols, not as

mathematical, networking, database, and graphics component libraries.

Second, Draco components provide multiple refinements of a component into other domains – not necessarily executable source code. This provides three features: high-level domain specific optimization, variety in implementation goals, and variety in implementation architectures. As an example, a domain for describing communications protocols may: 1) optimize a particular domain language description; 2) use the same improved description to produce program code (e.g., C), code & hardware (e.g., C & VHDL), hand simulation educational demo, classical simulation input (e.g., SimScriptII), formal theory deadlock analysis input (e.g., Promela), or graphic diagrams; and 3) implement the code and hardware oriented results as parallel code, inline code, threaded code, or threaded code interpreters. When other DA implementation techniques discuss "the generator", there is usually an implicit target of program code. The target is not always just program code, although it should always be a requirement.

Third, the use of *conditions* and *assertions* on the various refinements of a component allow us to address the important *reusability questions.* Given a system description (e.g., protocol description), a set of domains, and a set of target domains (e.g., C & VHDL) the reusabilty questions are:

1. Can the description be refined to only the targets?
2. If so, what is a possible implementation?
3. If not, what system description changes, additional domains or additional refinements are needed?

These questions are important in a complex domain hierarchy because they directly address the issue of scale supported by composing domains. From a formal theory standpoint, questions 1 and 2 are decidable (with a high complexity in the general case) only if a limited condition/assertion language and a limited refinement mechanism are used [5]. What can be done about question 3 is unknown but important. This points out some key features of the Draco approach: the use of a restricted refinement mechanism, the limiting of some domain parts to be intradomain only, and the use of restricted power condition and assertion expressions. These decisions represent a conscious tradeoff between generative mechanism power and the ability to analyze what the mechanism will do given a particular problem to refine. This is unimportant if a DA generator takes specifications in a single domain and generates directly into code forms, because there is no composition of domains. However, as many domains are added to the domain hierarchy, the ability to analyze the domain interactions as handled by the generative mechanism becomes crucial to the development of refinement (implementation) strategies. With all DA generation techniques, the reader should consider:

1. Is the mechanism general? Can a new domain be added without reprogramming the mechanism?
2. Does the generative mechanism support domain composition? Do new domains reuse each other and not displace a single domain?
3. Is the mechanism scaleable to programming in the large? Can the mechanism be analyzed in action on large problems using a large set of domains?

The Draco approach provides the first two and is weak on scaling, although some work looks promising [6].

3. Conclusions

Automatic Programming research in the 1970s rated their techniques using a *power function*. The power function of a technique is the ratio of effort required to develop a system using the technique over the effort required to develop the same system using conventional techniques. Software Reuse recognizes that large gains in the power function are achieved by reusing system artifacts. Domain Analysis recognizes that large gains in reuse are achieved by reusing as much analysis and design as possible. The Draco approach recognizes that large gains from DA are achieved by having the domains reuse each other.

The power function fails to recognize the expense of putting the technique in place. I believe we are on an evolutionary path from cheaply implemented component libraries through one domain generation to very expensive domain hierarchies. As more demands are put on software production and education, the costs will become justified.

4. References

[1] Royce, W. W., *Managing the Development of Large Software Systems*, reprinted in 9th Intl. Conference on Software Eng., pp. 328-338, March, 1987.

[2] Simon, H. A., The Sciences of the Artificial, MIT Press, 1969.

[3] Neighbors, J. M., *An Assessment of Reuse Technology after Ten Years*, 3rd International Conference on Software Reuse, pp. 6- 13, November 1994.

[4] Neighbors, J. M., *Draco: A Method for Engineering Reusable Software Systems*, Software Reusability, T. Biggerstaff & A. Perlis eds., Vol. 1., pp. 295-319, ACM Press, Addison-Wesley, 1989.

[5] Neighbors, J. M., S*oftware Construction Using Components,* Ph.D. diss., University of CA, Irvine, May 1980.

[6] Katz, M. D., Volper, D. J., *Constraint Propagation in Software Libraries of Transformation Systems*, Intl. Journal of Software Eng. and Knowledge Eng., Vol.2, No.3, September, 1992.