presented at ACM National Conference, December 1978

Validating the Utility of Abstraction Techniques

Mary Shaw, Gary Feldman, Robert Fitzgerald, Paul Hilfinger, Izumi Kimura, Ralph L. London, Jonathan Rosenberg, Wm. A. Wulf

A number of recent research efforts have been based on the hypothesis that encapsulation techniques, formal specification, and verification lead to significant improvements in program quality. As we gain experience with the language facilities produced by this research, we should attempt to validate that hypothesis. This paper poses this validation as the next major task in this area and outlines some ways to address it.

Key words: abstraction and representation; abstract data types; correctness; encapsulation; formal specification; modular decomposition; programming languages; programming methodology; proofs of correctness; types; validation; verification.

1 Introduction

Over the past few years, a large number of research efforts have addressed the problems of software cost and quality. These efforts have drawn on experience in such diverse areas as programming methodology, language design, program verification (proof), software packages, and operating systems. Although the efforts are diverse, they focus on one central theme. This theme is that software costs rise and quality declines because of the complexity of the software systems and that the complexity can be controlled by abstracting from the details of an implementation to the properties expected of the system.

The information that contributes to complexity may pertain to control or to data. Language facilities such as routines and macros have supported control abstractions in languages for many years, but facilities for data abstraction are a more recent development. Data abstraction mechanisms with language support for protecting the implementation are provided in Alphard [Hilfinger 78, Shaw 77, Wulf 76], CLU [Liskov 77], Euclid [Lampson 77], Gypsy [Ambler 77], Mesa [Geschke 77], Modula [Wirth 77], and the Steelman requirement [Steelman 78]. They typically include

Authors' addresses: M. Shaw, G. Feldman, R. Fitzgerald, P. Hilfinger, J. Rosenberg, W.A. Wulf, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213; R.L. London, USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, California 90291; I. Kimura, Carnegie-Mellon University and Department of Information Science, Tokyo Institute of Technology, Ookayama, Meguroku, Tokyo 152, Japan. This work was supported in part by the Defense Advanced Research Projects Agency under contracts DAHC 15-72-C-0308 and F44620-73-C-0074 (which is monitored by the Air Force Office of Scientific Research), and in part by the National Science Foundation under Grant MCS77-03883.

- an encapsulation facility (based on a scope rule) that allows several routines to share private definitions

- a concept of type that allows a programmer to augment the primitive types of the language, including a declaration mechanism that can interface nonprimitive type definitions to the underlying allocation and parameter-checking facilities

In addition to such language mechanisms, data type abstractions may be supported by formal specifications of the abstract properties desired [Guttag 78, Roubine 76, Wulf 76]. Verification techniques for these abstractions are based on Hoare's work [Hoare 72]. The use of abstract data types to simplify program organization and development is presumed to simplify the specification and verification tasks as well.

The proposition that encapsulation tools, formal specifications, and verification techniques support abstraction and, in turn, program development has been well received and even enthusiastically endorsed. We believe, however, that the methodology has yet to be proved effective in actual practice on problems of significant size. The remainder of this paper poses validation of that proposition as the next significant task in this research area.

It is essential for projects that investigate the utility of the data abstraction methodology to address all three aspects of that methodology. Programs that are not accompanied by verifications do not completely test the decomposition into abstract data types. Similarly, specifications and verifications of programs in an unimplemented language fail to test whether specifications are complete.

2 Experience with Verifying Programs Developed through Data Abstraction

Although some large programs have been formally verified and some programs have been developed using the ideas about modularity that support abstract data types, no significant, non-toy program has yet been developed and verified using these techniques. We will provide a representative survey of the current status based on published papers and reports. Our purpose is to establish our belief that there is not yet enough systematic information available to evaluate the techniques.

One of the largest programs that has been developed with abstract data types and completely verified is the symbol table of [London 78]. The program text, including specifications, is about 150 lines. Another symbol table example [Guttag 76] is specified using algebraic axioms for each of the several abstractions and also is completely verified. With the specifications and implementations, the size is about 100 lines. A message switching network, about 200 lines of code and specifications, has been completely through a system for incrementally designing and verifying programs [Moriconi 77].

The new CLU compiler [Liskov 78] is implemented in CLU and relies heavily on clusters and iterators. The compiler itself uses 21 data abstractions and the CLU system uses six. Of these, only one is parameterized. No formal specification or verification has been undertaken. The implementors' reactions are strongly positive; they feel that data abstractions have been essential to the work.

The Mesa users, primarily the implementers, have described their experiences with abstractions and with strict type checking, including rewriting the compiler in its own language [Geschke 77]. They note, "it is hard to overestimate the value of articulating abstractions, centralizing their definitions, and propagating them" [p. 543, column 1]. We expect other experiences to be reported in companion papers at this session.

The notions of abstraction and modularity also appear in concurrent programming. For example, the language Concurrent Pascal extends Pascal by adding concurrent processes and monitors [Brinch Hansen 77]. Brinch Hansen's book contains three example programs: a one-user operating system (1300 lines), a job stream system (1800 lines, including 700 lines from the first example), and a real time scheduler (400 lines). Staunstrup [Staunstrup 78] discusses the verification of the first example.

Another example involving concurrency is a network communication system [Wells 76] which resembles the IMP subnet of the ARPA net. The example, which incorporates a substantial amount of concurrency, consists of about 1500 lines of Gypsy specifications and about 1000 lines of Gypsy code [Good 77]. A large portion of this example, including all of the concurrency, has been verified. A related example [Horn 77] involves a little over 900 lines of Gypsy specifications for a secure communications network. Each of these examples, however, makes only minimal use of data abstractions.

The detailed description of the design of a secure operating system appears in [Neumann 77]. The specifications, written in Special [Roubine 76], describe a 14-level hierarchical structure that is reflected in the implementation and in the verifications. This report contains outlines of (1) rigorous proofs that the specifications satisfy the desired security assertions and (2) proofs that the implementation programs are consistent with the specifications.

Honeywell [Boebert 77] is conducting an evaluation of SRI's specification technology in Special [Roubine 76]. They have written specifications for portions of an existing program and checked them with the SRI tools, but they have apparently not generated code yet.

3 The Validation Problem

In order to achieve the broad goal of demonstrating that encapsulation techniques, formal specification, and verification do *in fact* lead to better programs at lower costs, we must pose some more specific goals and set forth some criteria for evaluating how well those goals are met. In this section we discuss several ways to do this.

The issues that must be addressed in order to validate the data abstraction methodology include the following:

- *Feasibility*: Can the methodology be applied to systems of realistic size? Is it even *possible* to specify the properties of a sizable system in terms of abstract data types, then develop the system as a set of modules that correspond to those abstract data types, and verify that the implementation meets the specification?

- *Completeness of specification and verification*: Can we write formal specifications for all interesting properties, or must some remain informal (e.g., in prose)? Can all the formally specified properties be verified? If not, what class(es) of properties cannot be specified or verified?

- *Quality of programs*: How do programs developed via this methodology compare with conventional programs in terms of speed, size, clarity, reliability, and maintainability?

- *Life-cycle costs*: Are the total costs of these programs lower than for conventional programs? The total cost includes components arising from design, development, maintenance, and subsequent enhancement.

- *Reusability of programs*: Can abstractions developed for one application be reused in support of another application? In other words, can libraries of verified abstractions eventually reduce development costs, thus distributing some costs over more projects?

- *Implementability of languages:* Can high-quality code be produced for programs based on abstract data types? Language features such as generic definitions and strong typing, which are associated with abstract data types, pose new problems for compiler implementors.

It is difficult to design experiments to test how well abstraction techniques address these goals. As in any experiment involving programs of reasonable size, the amount of uncontrolled variation is sure to be large. In addition, many of the criteria are qualitative rather than quantitative. However, we believe that if projects are carefully selected and monitored, it will be possible to make concrete evaluations and draw responsible conclusions.

Projects that contribute to the validation of this methodology should demonstrate encapsulation, specification, *and* verification. Examples of suitable projects might include

- Reimplement an existing program. More specifically, formalize the desired properties of an existing program, reimplement it using data abstractions, and verify the correctness of the result. Compare the old and new programs along the dimensions of program quality mentioned above.

- Similarly, reimplement some parts of an existing program that may be more widely useful.

- Develop a set of verified abstractions to support a particular problem domain. (This could be regarded as a collection of specialized abstract data types.) Evaluate these abstractions in terms of their adequacy in the problem domain and how well the implementations remain hidden when the abstractions are used.

- "Shadow-bid" on a contract for a software system. Perform an independent implementation using abstract data types and compare both the development process and the resulting system to the project being "shadowed".

One could, of course, consider a large-scale experiment in which many implementation groups apply one of the two methodologies to a single problem. To perform a truly controlled experiment, it would be necessary to control for variation among individual programmers by doing many implementations, to monitor all development costs, to use all versions in operational situations, and so on. It is not practical to run such experiments for full-scale implementation projects.

4 The Alphard Effort

For Alphard, we have elected to approach the validation problem by reimplementing a conventionally-developed system. This will allow us to test the overall feasibility of the methodology and to make comparisons between the programs. We selected the following organization and ground rules for the project.

- Choose an existing program of medium size (15 - 20 pages of code).

- Write a specification based on this system. Although exact adherance to the design of the base system is not required, the specification must be close enough to permit comparisons.

- Reimplement the program completely, basing the implementation design on abstract data types. This implementation need not be faithful to the original program.

- Verify that the implementation satisfies the specifications.

- Compare program sizes and execution times quantitatively.

- Compare the formal specifications to conventional documentation in terms of size, completeness, and understandability.

- Evaluate clarity, maintainability, and the generality of components subjectively.

In order to compare a program developed in Alphard to a conventional program, we need a basis for comparison. We considered independent implementations of a given problem and reimplementation of a component of a running system. We rejected both possibilities because the conclusions of any evaluation would depend as much on the quality of the conventional program as on the quality of the Alphard program. We decided instead to reimplement a program that has been polished and fine-tuned for publication. The problem we have chosen is the text editor developed in [Kernighan 76, Chapter 6]. By selecting a published program as the basis for our validation effort, we hope to make the results more accessible and easier to interpret.

This task will not, of course, address all the dimensions of the validation task. For example, we have no metric for evaluating reliability or maintainability. Further, the experiment as described will not address long-term considerations such as modifiability and reusability of components. It is possible that these questions could be addressed in follow-on projects.

As of June 1978, a prose specification of the text editor has been completed. Since the specification is written in English, it is necessarily informal. However, it was developed carefully, and it is reasonably precise. The next steps, which are in progress, are recasting the specification in formal terms, selecting abstract data types for the initial system design, and implementing the Alphard language as defined in [Hilfinger 78].

5 Conclusion

Research on abstract data types has proceeded far enough to start demonstrating the practical uses of the approach. Although the initial results are promising, they are not conclusive. We have sketched some approaches to this task and discussed the experiment currently under way in Alphard. We look forward to seeing results from other research efforts.

We would like to express our appreciation to Donald Good for helpful comments on the status of current research and to Barbara Liskov for details on the CLU implementation.

6 Bibliography

[Ambler 77] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells, "Gypsy: A Language for Specification and Implementation of Verifiable Programs", *SIGPLAN Notices*, 12,3 (March 1977), pp.1-10.

[Boebert 77] W.E. Boebert, J.M. Kamrad, and E.R. Rang, "Analytic Validation. of Flight Software", *Honeywell Systems and Research Center Report*, 775RCG3, September 1977.

[Brinch Hansen 77] Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, 1977.

[Geschke 77] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite, "Early Experience with Mesa", *Communications of the ACM*, 20,8 (August 1978), pp.540-553.

[Good 77] Donald I. Good, "Constructing Verified and Reliable Communications Processing Systems" *Software Engineering Notes*, 2,5 (October 1977), pp.8-13.

[Guttag 76] John V. Guttag, Ellis Horowitz, and David R. Musser, "Abstract Data Types and Software Validation", *USC Information Sciences Institute Technical Report* ISI/RR-76-48, August 1976. Also *Communications of the ACM*, to appear.

[Guttag 78] John V. Guttag, Ellis Horowitz, and David R. Musser, "The Design of Data Type Specifications", in *Current Trends in Programming Methodology* (R.T. Yeh, ed), Prentice-Hall, 1978 (pp. 60-79).

[Hilfinger 78] Paul Hilfinger, Gary Feldman, Robert Fitzgerald, Izumi Kimura, Ralph L. London, KVS Prasad, VR Prasad, Jonathan Rosenberg, Mary Shaw, Wm. A. Wulf (editor), "(Preliminary) An Informal Definition of Alphard", *Carnegie-Mellon University Technical Report* CMU-CS-78-105, February 1978.

[Hoare 72] C.A.R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica*, 1, 4, 1972 (pp. 271-281).

[Horn 77] Gary R. Horn, "Specifications for a Secure Computer Communications Network", Master's thesis, University of Texas at Austin, October 1977.

[Kernighan 76] B.W. Kernighan and P.J. Plauger, *Software Tools*, Addison-Wesley, 1976.

[Lampson 77] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek, "Report on the Programming Language Euclid", *SIGPLAN Notices*, 12,2 (February 1977).

[Liskov 77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert, "Abstraction Mechanisms in CLU", *Communications of the ACM*, 20,8 (August 1978), pp.564-576.

[Liskov 78] B. Liskov, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder, "The CLU Reference Manual", *Computation Structures Group Memo* No. 161, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1978.

[London 78] Ralph L. London, Mary Shaw, and Wm. A. Wulf, "Abstraction and Verification in Alphard: A Symbol Table Example", *Constructing Quality Software*, P.G. Hibbard and S.A. Schuman (eds.), North-Holland, 1978 (pp.319-351).

[Moriconi 77] Mark S. Moriconi, "A System for Incrementally Designing and Verifying Programs", Volumes 1 and 2, *USC Information Sciences Institute Technical Reports* ISI/RR-77-65 and 66, November 1977. Also Ph.D. thesis, University of Texas at Austin, 1977.

[Neumann 77] Peter G. Neumann, Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson, "A Provably Secure Operating System: The System, its Applications, and Proofs", *SRI International Project* 4332 Final Report, February 1977.

[Roubine 76] O. Roubine and L. Robinson, "Special (SPECIfication and Assertion Language): Reference Manual", *SRI International Memo*, August 1976.

[Shaw 77] Mary Shaw, Wm. A. Wulf, and Ralph L. London, "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators", *Communications of the ACM*, 20,8 (August, 1977), pp.553-564.

[Staunstrup 78] J. Staunstrup, "Specification, Verification, and Implementation of Concurrent Programs", Ph.D. thesis, University of Southern California, 1978.

[Steelman 78] Department of Defense Requirements for High Order Computer Programming Languages, "Steelman", June 1978.

[Wells 76] Robert E. Wells, "The Specification and Implementation of a Verifiable Communication System", Master's thesis, University of Texas at Austin, December 1976. Also Technical Report ICSCA-CMP-4.

[Wirth 77] Niklaus Wirth, "Modula: A Language for Modular Programming", *Software -- Practice and Experience*, 7,1 (January 1977), pp.3-35.

[Wulf 76] Wm. A. Wulf, Ralph L. London, and Mary Shaw, "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Transactions on Software Engineering*, SE-2,4 (December 1976), pp.253-265.

Table of Contents