Reverse Reverse-Engineering

Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
+1 404 894 8450
spencer@cc.gatech.edu

Terry Shikano
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
+1 404 894-7553
shikano@cc.gatech.edu

R. E. Kurt Stirewalt
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
+1 517 355-2359
stire@cse.msu.edu

ABSTRACT

Reverse engineering a program constructs a high-level representation suitable for various software development purposes such as documentation or reengineering. Unfortunately however, there are no established guidelines to assess the adequacy of such a representation. We propose two such criteria, completeness and accuracy, and show how they can be determined during the course of *reversing* the representation. A representation is successfully reversed when it is given as input to a suitable code generator, and a program equivalent to the original is produced. To explore this idea, we reverse engineer a small but complex numerical application, represent our understanding using algebraic specifications, and then use a code generator to produce code from the specification. We discuss the strengths and weaknesses of the approach as well as alternative approaches to reverse engineering adequacy.

Keywords

Reverse engineering, algebraic specification, code generation, adequate representation

"Living backwards!" Alice repeated in great astonishment. "I never heard of such a thing!"

— Lewis Carroll

1 INTRODUCTION

Reverse engineering is a powerful method for comprehending a software system. It produces a high-level representation of a program that is useful in software maintenance tasks such debugging, reengineering or documentation. Unfortunately, this definition is not helpful to software engineers and their managers in planning and managing reverse engineering efforts. In particular, it does not give any guidance to determining the necessary completeness and accuracy of the resulting representation. That is, it is hard to know when the understanding gained adequately represents the original program. The research question we explore is *how do we know if a reverse engineering effort has produced an adequate representation of a program.* Our answer is that a program representation produced by reverse engineering is adequate if it is sufficiently complete and accurate that an automated tool is capable of reconstructing a program equivalent to the original from it.

We apply two key insights to explore adequate reverse-engineering representations. Our first insight comes from examining the idea of *reversing* the reverse engineering process; that is, in taking the representation that results from reverse engineering and using it to reconstruct a program. In order to reduce variation and uncertainty in the process, we want the reconstruction to be done automatically. If we are able to do these two tasks, representation and generation, then we have an operational means for determining the completeness of our effort. The second insight concerns the quality of the representation; that is, how do we ensure that our representation provides useful insight into the program. Here we use a model of the program's application domain as an external standard against which we compare the representation as it is built up during reverse engineering. A domain model provides a set of expectations for constructs in the program and how they relate to each other. Comparing understanding gained while reverse engineering a program to expectations provided by the domain model encourages an accurate program representation.

We illustrate the application of our insights by reverse engineering a small but complex numerical program using algebraic specifications as a notation for the representation. A code generator uses the representation to reconstruct an equivalent version. Based upon this case study, we critique the strengths and weakness of the approach.

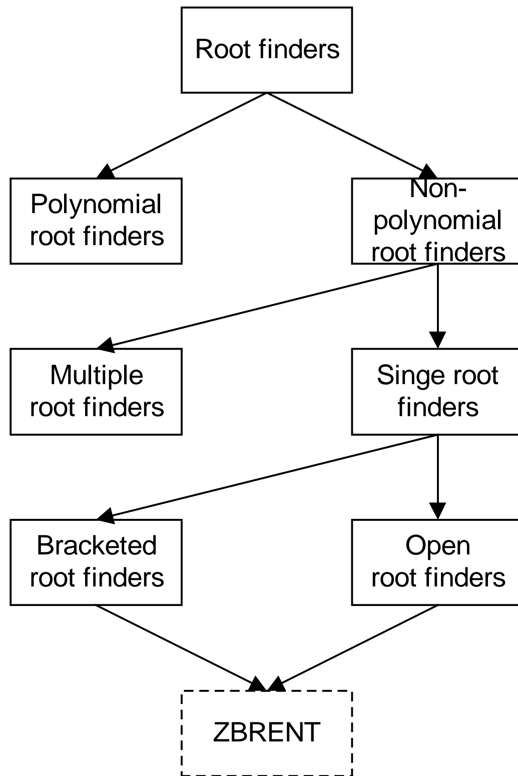

Figure 1: Lineage for ZBRENT

2 ADEQUACY

Wyuker pointed out the relevance of adequacy to the testing of software [12,13]. In particular, successfully executing an adequate set of tests gives management at least one indication that sufficient testing has been performed. We would like an analogous indication to help manage the reverse-engineering process.

The role of a reverse engineer is to understand a program and to express that understanding using a well-defined representation mechanism. To do this, the reverse engineer must answer two kinds of questions: *what* questions and *how* questions [8]. *What* questions concern the goals and requirements of the program; whereas *how* questions concern the design decisions made by the original developers and how these decisions are manifested in the code [11]. The adequacy of a representation indicates whether the representation completely and accurately expresses *what* and *how* information.

Adequacy can be viewed either extrinsically or intrinsically. Extrinsic criteria define the adequacy of an artifact in terms of how it is used. In the case of testing, reliability is an example of an extrinsic criterion. For example, if 99.9% reliability is used as a testing criterion, then it is unlikely that users will actually experience a failure. Intrinsic criteria refer to properties that can be determined directly from an artifact itself. For example, statement coverage, which indicates that each line of code in a software system has been tested, is an example of an intrinsic testing criterion.

Unfortunately, extrinsic criteria pose a practical problem for management because they cannot be assessed until the artifact is used. In the case of reverse engineering, the uses to which a representation is put are key extrinsic determinants, but we may not be able to anticipate those uses at the time we are performing the reverse engineering. Hence, we would like intrinsic criteria to help manage the reverse engineering process.

We have looked at two intrinsic adequacy criteria: completeness and accuracy. A representation is deemed *completeness adequate* (or *complete*) if it is capable of being automatically used to reconstruct a program equivalent to the original. In addition to completeness, we need an adequacy criterion to indicate the extent to which a representation elucidates the design of the software and the connection between the software and the goals it was designed to accomplish. To the extent that a representation provides this insight, we call it *accurate*.

Our primary contribution is a framework for understanding how to objectively assess completeness and accuracy. For completeness, we require an automated code generator to be able to process the representation and transform it into a program that is equivalent to the program being reverse engineered. We judge two programs equivalent if one produces the same result as the other when run on a set of test data. Obviously, the strength of this judgement depends on the (testing) adequacy of the test suite being used.

To assess accuracy, we make use of a model of a program's application domain. Application domains are mature and cohesive sets of applications characterized by a common vocabulary, literature and solution architecture [1]. Domain models express the important concepts in the domain and relationships among them. As such, they provide a vocabulary with which *what* information can be communicated. For our purposes, a representation is considered accurate, if it faithfully articulates concepts and relationships in the program's domain model and connects these domain concepts to the program code in a way that makes design decisions explicit.

3 CASE STUDY

To explore the issue of adequate representations, we reverse engineered a numeric application called ZBRENT, written in the C programming language. The particular application domain was that of finding the root of a real-valued function. A domain model was obtained from textbooks. Then the SLANG algebraic specification language was used to represent both the

domain model and the algorithm. SLANG is part of the Specware tool from Kestrel Institute. We made use of the SLANG code generator in Specware to produce an executable C++ program for ZBRENT, which was then tested against the original program on a set of test functions. We then assessed the completeness and accuracy of the resulting representation.

Root Finding

Finding the root of a nonlinear equation is a well-understood problem in numerical analysis [4,5]. Programs have existed for many years to robustly and efficiently compute roots [9]. Consequently, root finding qualifies as an application domain suitable for domain-based program understanding [10]. That is, there is a sufficiently large collection of programs for finding roots that we can identify common characteristics and use them as expectations to guide the reverse engineering process.

As illustrated in **Figure 1**, at the top level, the root-finding application domain can be partitioned into polynomial and non-polynomial families of algorithms. Within the latter, a further distinction exists between algorithms capable of finding multiple roots and those capable of dealing with a single root only. For the purposes of this paper, we are concerned with the latter class. Within single-root-finders, a final distinction is made between those guaranteed to converge—bracketed root finders—and those, presumably more efficient ones, that do not—open root finders.

Single-root, non-polynomial root finders work in the following way, as illustrated in **Figure 2**. Input consists of a subprogram `(f)` capable of computing the functional value at a given real value `(x)` and an initial estimate of the root in the form of a containing subinterval of the real line, denoted by its end points. Functional evaluation is typically expensive, so root-finding algorithms try to reduce the number of calls to the argument subprogram.

Root finding proceeds by selecting a trial point within the current interval thereby partitioning the interval into two pieces, determining the piece containing the root, creating a new, refined interval using the chosen piece, and iterating. In the figure, interval end points are indicated with dashed lines. Increasing subscripts on interval names denote the order of refinement. For example, interval i_2 is refined by the smaller interval i_3. Robust root-finding algorithms have the property that the interval gets smaller on every iteration. Moreover, a stopping criterion determines whether sufficient progress has been made to warrant continuing the process.

Algorithms of this sort can be categorized by specifying the method by which a refined interval is chosen and by the stopping criterion. Variations of the first

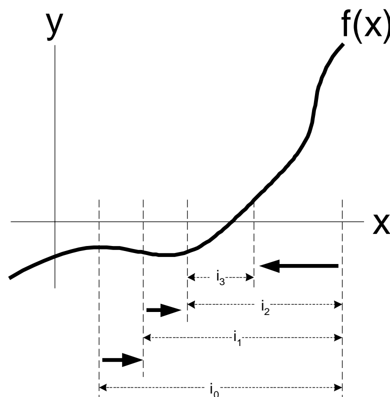

Figure 2: Iterative Interval Shrinkage

sort include bisection (Bolzano's method), linear interpolation (Regula Falsi), inverse quadratic interpolation (Mueller's method), Aiken's Delta Squared, Newton-Raphson, and secant. Variations of the second sort include stopping when the functional value is sufficiently close to zero, stopping when the interval width is sufficiently narrow, and stopping after a fixed number of iterations. Of course, multiple methods of each sort can be combined to improve robustness or efficiency.

ZBRENT

Our case study examined a particular root finding algorithm called ZBRENT. The algorithm was first published by Decker in 1969 and improved by Brent in 1973. It combines several of the variations described in the previous section in order to improve efficiency and robustness. We have chosen the Brent variation, as taken from [9], as the subject of our case study, for several reasons.

- It is written in the C language. This allows us to more directly compare it with the output of our chosen code generator, which produces C++ code.
- It features all three choices of stopping criteria.
- It features three interval shrinkage methods: bisection, secant, and inverse quadratic interpolation.

As a consequence of the number of variations, the code is complex and difficult to follow, making it a good candidate for reverse engineering. The actual code consists of a single function of 101 lines of which three are preprocessor lines, three are whole-line comments, sixteen are blank lines, eight are lines containing a single opening or closing brace, and nineteen are declarations, leaving 52 executable statements.

Algebraic Specification

We have chosen algebraic specification as a notation for representing the results of the reverse engineering of ZBRENT. Algebraic specification is an application of

formal methods to the problem of precisely specifying the behavior of a software system. Specifications are decomposed using *sorts* (data types) and the operations that manipulate them. Operations are in turn defined via *axioms* (sets of equations) indicating how the value computed by one sequence of operations equates to that computed by another.

If the equational definitions in an algebraic specification are suitably constrained in format, they may be interpreted operationally as a set of rewrite rules, thereby enabling compilation into a traditional programming language. We have used an algebraic specification language, called SLANG, which comes as part of the Specware tool suite from Kestrel Institute.

Specware

Specware is a tool for developing software from specifications [7]. The particular notation that it uses, called SLANG, is algebraic in nature, and the accompanying formality enables powerful manipulations to be performed including generation of code guaranteed to correctly implement the specifications. We used SLANG to represent the results of the reverse engineering of ZBRENT.

As an example, the SLANG code shown in **Figure 3** denotes the specification of an interval, suitable for use in building a domain model for root finders. This specification for `INTERVAL` (lines 1-20) describes a sort called an `Interval` (line 3) making use of a previously defined specification called `EXTENDED-REAL` (line 2) that is used to model x-axis values. The structure of an `Interval` is defined with a sort axiom (line 4) as being the Cartesian product of two `Reals`. In addition to sorts, specifications define operations and constants. In `INTERVAL` there are two operations defined (`mid-point` and `make-interval`) but no constants. Operation definitions consist of a signature and one or more axioms. For example, the signature for `mid-point` (lines 6) indicates that it takes as input an `Interval` and produces as output a `Real`. The single axiom defining `mid-point` (lines 8-9) asserts that the output value it produces, when given as input the `Interval` constructed from values `a` and `b`, is equal to the value produced when the `half` operation is composed with the results of operating on `a` and `b` with the `plus` operation. In a similar manner, the `make-interval` operation is defined on lines 12-16. Finally, lines 18-19 indicate that `the make-interval` operation provides a way in which new `Intervals` may be instantiated.

SLANG Support For Adequate Representations

In Specware, specifications are actual data values that can be manipulated by high-level operators called *morphisms*. In particular, SLANG provides three morphisms: *import,* for including one specification inside

```
( 1) spec INTERVAL is
( 2)   import EXTENDED-REAL
( 3)   sort Interval
( 4)   sort-axiom Interval = Real, Real
( 5)
( 6)   op mid-point : Interval -> Real
( 7)   definition of mid-point is
( 8)     axiom mid-point(a, b) =
( 9)       half(plus(a, b))
(10)   end-definition
(11)
(12)   op make-interval : Real, Real ->
(13)     Interval
(14)   definition of make-interval is
(15)     axiom make-interval(a,b) = (a,b)
(16)   end-definition
(17)
(18)  constructors { make-interval }
(19)    construct Interval
(20) end-spec
```

Figure 3: SLANG INTERVAL Specification

another; *translate,* for renaming the sorts and operations of a specification; and *colimit,* for combining specifications in a structured way. By writing atomic specifications and then using the morphisms to combine them, complex systems can be cleanly modeled.

We make use of one other Specware feature, called an *interpretation*, which is Specware's way to formalize design refinements. Refinements relate abstract domain-model concepts to executable code. Operationally, an interpretation demonstrates how the sorts and operations in one specification are implemented using sorts and operations in another specification at a lower level of abstraction. Interpretations are not morphisms, but rather are an idiomatic collection of morphisms and a special intermediate specification called a *mediator*. Briefly, a mediator defines operations in the source specification using axioms that reference operations in the target specification.

Mediators play a central role in our approach because they explicate design decisions that explicitly relate domain concepts to implementation concepts via interpretations. Interpretations force a reverse engineer to understand and demonstrate how domain concepts are connected to implementation concepts. By requiring the use of interpretations to connect domain and implementation concepts, we make accuracy objective and add rigor to the understanding process.

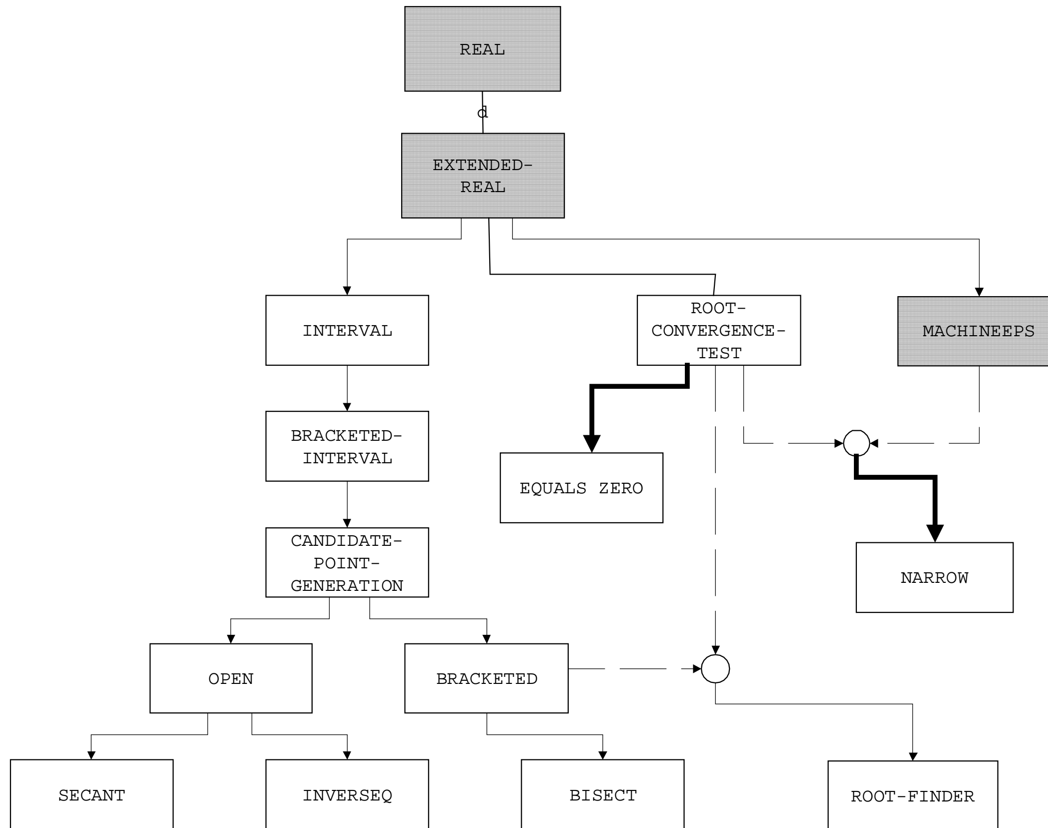

Figure 4: Root Finding Domain Model

Process

The reverse engineering of ZBRENT is expressed in a three-part SLANG program. The first part comprises a domain model constructed by reading descriptions in books and papers on root finding and articulating them in SLANG. Next, we expressed the ZBRENT source code as a set of SLANG operation definitions. Essentially, these implementation operation definitions just recast the code in a slightly more functional manner such as might be produced in writing an SML version of ZBRENT. We then embarked on an iterative process of finding SLANG interpretations to connect the implementation operations to domain concepts. Often, to introduce an interpretation required refactoring the implementation specification. We only allowed changes that resulted in a complete representation, i.e., one for which Specware could generate a program that was testing equivalent to the original C code for ZBRENT. We stopped this process when we were able to connect every implementation specification to the appropriate domain specification.

4 RESULTS

The Root-Finding Specification

Figure 4 depicts the root-finding domain model as a set of related Specware specifications. Boxes in this figure denote specifications, which represent different concepts and relationships in the domain model. Filled boxes are not actually part of the domain model but provide resources to it from other domains. In particular, `REAL` is the specification for real numbers, and, in `EXTENDED-REAL,` we add several useful utility operations. The `MACHINE-EPS` specification can be thought of as belonging to a separate domain describing the properties of floating-point arithmetic on a physical machine. In this case, it provides the definition of a small constant value useful in determining when a floating-point computation is unable to contribute significantly to a computation.

The root finding domain model proper consists of twelve specifications, which can be further subdivided into three pieces describing the iterative root finding process (the bottom-right box), convergence (the rightmost three unfilled boxes), interval shrinking (the remain unfilled boxes). The specific roles of the specifications are summarized in the following list.

- `INTERVAL`: the data structure holding interval end points;

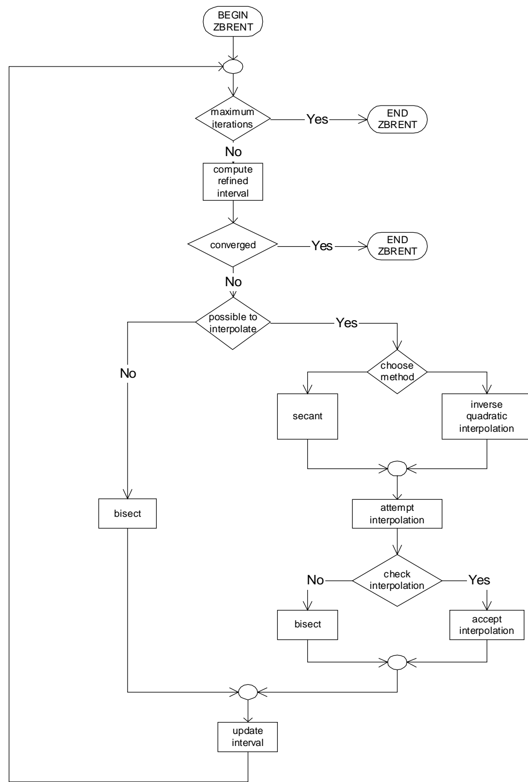

Figure 5: Abstract Flow Chart for ZBRENT

- `BRACKETED-INTERVAL`: an interval guaranteed to hold a root;
- `CANDIDATE-POINT-GENERATION`: the process of generating a candidate end point for a refined interval;
- `OPEN`: the generation of points not guaranteed to be contained in the current interval;
- `SECANT`: the secant method for generating a new candidate end point;
- `INVERSEQ`: the inverse quadratic interpolation method for generating a new candidate end point;
- `BRACKETED`: the generation of points guaranteed to be contained in the current interval;
- `BISECT`: the bisection method for generating a new candidate end point;
- `ROOT-CONVERGENCE-TEST`: the properties of any convergence test;
- `EQUALS-ZERO`: a convergence test in which a root is exactly found;
- `NARROW`: a convergence test in which the current interval has become too small for further progress;
- `ROOT-FINDER`: the process of continually shrinking an interval until a convergence test is passed.

In addition to the twelve domain specifications, **Figure 4** also illustrates several kinds of morphisms. The line from `REAL` to `EXTENDED-REAL` is labeled with the letter *d* to denote what is called a *definitional extension*. `EXTENDED-REAL` can be thought of as a macro package providing a set of abbreviations shortening the expression of various composite `REAL` operations. An unadorned line in the figure denotes *an import morphism* indicating the textual inclusion of one specification within another. This is normally used to build a new specification that makes use of the features of an old one. For example, the `INTERVAL` specification needs to import from the `REAL` specification because the interval end points are real numbers. The bold lines in the figure correspond to *translate* morphisms in which one or more imported specification elements have been renamed. In this case, the renaming allows the `ROOT-FINDER` specification to be written using an abstract convergence test which, in the ZBRENT algorithm, is refined by the disjunction of the two concrete tests specified in the figure. Finally, and most interestingly are the two *colimit morphisms* denoted by dashed lines ending in a small circle. A colimit is a shared union of the two source specifications. It is used within algebraic specifications for a variety of purposes including, in this case, the composition of multiple source specifications into a target specification.

The ZBRENT Specification

In addition to the root-finding domain model, a specification must be given for the ZBRENT algorithm itself. **Figure 5** contains a high-level flow chart for the C version of ZBRENT. The outer loop wraps the shrinking process and ensures termination by counting iterations. Within this loop, it is possible to terminate successfully with a root if either the functional value at an interval end point is zero or the interval itself has grown too narrow. If termination is not warranted, a check is made to see whether interpolation is promising, and, if so, either secant or inverse quadratic interpolation is chosen. As neither of these is guaranteed to produce a bracketed value, a subsequent check must be made. If any of these tests fail, then bisection is used. Finally, the interval is updated with the appropriately chosen subinterval.

Not shown in the flow chart are various optimizations—memoization of functional values to avoid recomputation, beginning interpolation from the end point whose functional value is smallest in magnitude, and remembering previous interval end points to speed interpolation.

The SLANG specification of the ZBRENT algorithm consists of a rendering of the flow chart boxes and optimizations with axioms. This process is similar to writ-

ing a functional program in a language such as SML or Haskell. Usually, the major effort is in converting occurrences of the assignment statement, for which there is no equivalent in SLANG. Instead the value assigned is passed as a argument to all subsequent computations requiring it.

Interpretations

The SLANG representation of the ZBRENT algorithm is complete in the sense that code equivalent to the original can be generated from it. However, it sheds no light on how the algorithm accomplishes the goal of finding a root. We use SLANG interpretations for this purpose. In particular, an interpretation indicates precisely how an abstract domain concept is manifested in the algorithm. To the extent that each aspect of the algorithm specification is tied to a domain concept, the combined domain and algorithm specification is judged accurate.

The Generated Code

Specware is capable of generating code from SLANG specifications using either Lisp or C++ as a target language. We chose C++ to facilitate comparison with ZBRENT. Specware's code generator does not take advantage of any of C++'s object-oriented feature, further easing the comparison. The three parts of the ZBRENT specification (domain model, algorithm, and interpretation) required respectively 147, 303, and 75 lines of SLANG code. The total code generated for ZBRENT consists of 624 lines, of which two are preprocessor lines, six are whole line comments, 124 are blank lines, and 47 are declarations, leaving 445 executable statements.

5 EVALUATION

We proposed a definition of the adequacy of a reverse-engineering representation based on completeness and accuracy. We then discussed how to objectively measure these criteria by using SLANG to represent the results of reverse engineering, checking completeness with the aid of the Specware code generator and validating accuracy using a domain model. Having demonstrated proof of concept, we now subject our results to a critical analysis.

Upon reflection, we identified four criticisms that might be raised:

- Are our methods for assessing completeness and accuracy truly objective?
- Is our definition of intrinsic representational adequacy better than other definitions?
- Given industry's resistance to formal methods, is our approach practical?
- Would our methods fare well in other application and solution domains?

We address each now in turn.

Objectivity of our Methods

Our methods for assessing completeness and accuracy are objective. Completeness can be assessed objectively through the use of the Specware code generator and a suite of test cases. The code generator automatically assembles an executable program from the algebraic specifications. This generated program is then compiled and run against every test in the test suite. If the generated program behaves differently than the original program on any one of these tests, then the programs are not equivalent. Our completeness criterion requires the programs to be indistinguishable by the test suite. Because the code is generated from the specifications without manual intervention, and because a program either passes or fails a test, our completeness criterion is objective.

Like completeness, our accuracy criterion is also objective. The objective arbiter in our method is the application domain model, which must reflect every domain concept that is detected during the process of representing design decision seen in the code. In the ZBRENT example, we used a domain model of root finding and, to a much lesser extent, domain models that describe machine arithmetic and real numbers. By having to identify an interpretation that connects every line of code to the salient domain concepts, and by virtue of a fixed, external domain model, we claim that our accuracy criterion is objective.

Other Adequacy Criteria

To our knowledge, our definition of representation adequacy in terms of completeness and accuracy is the first intrinsic definition of representation adequacy to be proposed. Objectivity is required of intrinsic methods, which by definition do not rely on external validation. If, over time, researchers develop new criteria to compete with completeness and accuracy, and if these new criteria can be assessed objectively, then it would be possible to set up empirical studies that assess them. Such studies would compare representations arrived at through the competing criteria against extrinsic factors.

Our choice of algebraic specification in general and SLANG and Specware in particular were motivated by our desire for an objective, intrinsic measure of representation adequacy. Clearly, however, the choice of notation influences the larger software-maintenance task for which the representation is used. Our results to date do not support any claims about the extrinsic adequacy of a representation that we deem intrinsically adequate.

Practicality

Formal methods have not gained wide acceptance in industry because they are often perceived as being difficult to learn and to apply [14]. SLANG is no exception: The authors had to invest a lot of effort to learn

how to use the notation and tools before we were able to craft a representation with the precision and detail required by the case study. However, because reverse engineering is known to be an expensive activity, we believe the benefits of an objective adequacy assessment outweigh the costs involved with using formal methods.

Beyond the general argument over formal methods, we were able to assess SLANG as a notation for representing the results of reverse engineering. The notation was particularly useful for experimenting with different conjectures about the relationship between *what* and *how* information. This experimentation was supported by the ability to automatically generate code and the powerful mechanisms for factoring and relating specifications. For example, in the early stages of our case study, we were able to separate implementation concerns into specifications with meaningful abstract names, even though the operations defined in these specifications were not yet at the appropriate level of abstraction. Typically, these early specifications are ugly and riddled with implementation detail, but the intended factoring and separation can still be expressed, and correct code can still be generated. Moreover, in SLANG, specifications that are riddled with implementation detail are painfully ugly; so in some sense, the notation provides aesthetic feedback to encourage factoring and compositional understanding.

The only drawback we experienced in using SLANG concerned a lack of familiar devices that we had grown to expect having used other modeling notations. Features in the algebraic paradigm are more abstract and declarative than those found in other modeling paradigms, specifically object-oriented modeling. Object-oriented modeling directly supports a rich set of abstraction mechanisms, such as encapsulation, identity, inheritance, and polymorphism. To become proficient in one modeling paradigm requires a methodological commitment to think in terms of that paradigm. Even though SLANG does not support these concepts, we found them difficult to do without when we were modeling ZBRENT. In fact, we often inadvertently tried to simulate them in SLANG. Often this thinking resulted in awkward specifications.

Application to Other Domains

Our results demonstrate that SLANG helps to elucidate the design of a program that solves a mathematical problem (root finding) in a procedural language (C). There are, of course, at least two dimensions of variation against which our methods must be validated for their ability to elucidate design decisions. The two dimensions are different application domains (e.g., control systems, decision support, etc.), and different design paradigms (e.g., object-oriented, real-time, etc.). Concerning the former, it is difficult to predict how a method will fare when it is applied to a different application domain. We intend to address this question in our future work.

On the other hand, we can make several observations about different design paradigms, at least with regard to our current use of algebraic specification and Specware. For example, it is difficult in algebraic specification to model state. Consequently, programs that exhibit a high degree of encapsulation and linked data might be difficult to generate using our approach. Even though Specware has a C++ code generator, we discovered that the generated C++ does not exploit any object-oriented features of the language, such as inheritance and polymorphism. We are exploring the extent to which this limitation is a necessary artifact of algebraic specification or if, in fact, it is just a current limitation in the Specware code generator.

In addition to difficulty with state and encapsulation, non-functional concerns, such as performance and resource utilization, are difficult to model using algebraic specification. In fact, such concerns are difficult to model in most modeling notations. Fortunately, in our experience with legacy systems, functional concerns far outweigh the non-functional concerns, and most systems are implemented in standard procedural languages, such as Fortran, Cobol, and C.

6 RELATED WORK

We are not aware of any other work concerning the adequacy of reverse engineering representations. However, several other groups have explored the use of formal methods to support reverse engineering. Perhaps the work most closely related to ours is that of Basili and Mills [2] in which they examine a program called ZEROIN, a Fortran implementation of ZBRENT. Basili and Mills are interested in applying ideas from structured programming and program correctness proofs to the understanding and annotation of computer programs. In particular, they construct a *prime-program decomposition* of a program's control flow graph, define *program functions* for each prime program, build a data reference table describing the accesses to and alterations of each program variable, and then synthesize a program correctness proof demonstrating exactly how the program accomplishes its goal.

Using this approach, the authors are able to gain an understanding of ZEROIN. The approach helps them organize and document their analysis and provides a completeness criterion. That is, they know that they have to keep working until they complete their proof. Hence, it gives them a way of knowing when their understanding is deep enough. Concerning accuracy, however, they make no direct comment on the extent to which their process relates design decisions to program

goals.

Hayes describes work similar to ours undertaken as a collaboration between IBM and Oxford University [6]. From the point of view of Oxford, the purpose of the described research was to explore the scaling of formal methods to industrial settings. From the point of view of IBM, the purpose was to gain a greater understanding of its Customer Information Control System (CICS) in support of its reimplementation and enhancement.

The specific effort undertaken was to mathematically model several CICS modules using the Z specification language augmented by informal, English language text describing the purposes of the modeled constructs. Hayes discusses problems that arose during the specification process itself, including communication difficulties between the mathematical modelers and the implementation experts, obtaining the right level of abstractness in the specification, and the exclusion of several aspects of the system, such as parallelism and distribution, that went beyond the capabilities of the modeling approach used.

We are interested in producing an adequate representation; that is, one that is complete and accurate. The Oxford models were complete in the sense that they described the behavior of the modules to the extent described in the manuals while incomplete in ignoring possible interactions due to parallelism and distribution. However, the comparison with the manual is inherently informal in contrast to our operational completeness test.

Concerning accuracy, Hayes asserts that the right level of abstraction occurs when the specification focuses on the primary purpose of a module while avoiding implementation details. This roughly corresponds with our guideline that suggests that the right level of accuracy occurs when there is a clear mapping between the generic domain concepts and the specific algorithm details.

Another related project was REDO [3], a part of the ESPRIT II European collaboration. A variety of activities were undertaken in REDO including the invention of a language, called UNIFORM, intermediate between code and specifications, an application of weakest-precondition semantics to the verification of programs, the definition of the Z++ extension to Z, an exploration of decompilation including the development of a decompiler compiler, and the activity closest to ours, the reverse engineering of existing applications.

The REDO reverse engineering process consisted of three stages: compilation from Cobol to UNIFORM, abstraction to a functional form, and simplification using normalizing transformations. The REDO work builds on the Basili and Mills approach but goes beyond it by detecting and characterizing important objects in the code corresponding to, for example, files, arrays and reports. However, there is no discussion of either completeness or accuracy.

7 CONCLUSION

For reverse engineering to become a routine part of the software development life cycle, managers must understand its costs and benefits. The cost of reverse engineering equates to the effort required to produce a high-level representation for a program. But because such representations vary greatly in detail and scope, it is necessary to have a precise definition of what such representations entail. Once such a definition exists, then data can be collected relating the effort required to produce a representation to the size of the program being analyzed. This data, in turn, can be used to better schedule the required effort.

The benefit of a reverse engineering effort correlates to the quality of the representation produced. In particular, high-quality representations describe not only the functioning of a program but also how that functioning accomplishes the program’s purpose. A definition of reverse-engineering representation that specifies this association permits managers to better relate the representation to its uses in subsequent software maintenance tasks.

We have proposed a definition for an adequate reverse engineering representation in terms of its completeness and accuracy, which, in turn, enable better understanding or reverse engineering costs and benefits.

ACKNOWLEDGEMENTS

We appreciate the support provided by the National Science Foundation (CCR-9708913). We would also like to acknowledge the help provided by the Specware team at the Kestrel Institute, particularly Jim McDonald, Dusko Pavlovic and Doug Smith. Our colleague on this project was Linda Wills. We benefited greatly in our discussions with her.

REFERENCES

1. Guillermo Arango and Rubén Prieto-Díaz. *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, (1991).
2. Victor R. Basili and Harlan D. Mills. Understanding and Documenting Programs. *IEEE Transactions on Software Engineering,* SE-8(3):270-283, (May 1982).
3. Jonathan P. Bowen, Peter T. Breuer, and Kevin C. Lano. The REDO Project: Final Report. Oxford University Computing Laboratory, PRG-TR-23-91, (1991).

4. G. Dalhquist and Å. Björck. Nonlinear Equations. *Numerical Methods,* Chapter 6, Prentice-Hall, (1974).

5. G. E. Forsythe, M. A. Malcolm, and C. B. Moler. Solution of Nonlinear Equations. *Computer Methods for Mathematical Computations.* Chapter 8, Prentice-Hall, (1977).

6. Ian J. Hayes. Applying Formal Specification to Software Development in Industry. *IEEE Transactions on Software Engineering.* SE-11(2):169-178, (February 1985).

7. Kestrel Institute *Specware User Guide,* Version 2.0.3, (March 1998).

8. S. Letovsky. *Plan Analysis of Programs.* Ph.D. Thesis, Yale University, (1988).

9. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Root Finding and Nonlinear Sets of Equations. *Numerical Recipes in C, The Art of Scientific Computing, Second Edition,* Chapter 9, Cambridge University Press, (1992).

10. S. Rugaber. The Use of Domain Knowledge in Program Understanding. *Annals of Software Engineering,* 9:143-192, (2000).

11. Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr. Recognizing Design Decisions in Programs. *IEEE Software,* 7(1):46-54, (January 1990).

12. Elaine J. Weyuker. Axiomatizing Software Test Date Adequacy. *IEEE Transactions on Software Engineering,* SE-12(12):1128-1138, (December 1986).

13. Elaine J. Weyuker. The Evaluation of Program-Based Software Test Date Adequacy Criteria. *Communications of the ACM,* 31(6):668-675, (June 1988).

14. Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer,* 10, (September 1990).