Model-Driven Reverse Engineering

Spencer Rugaber, *Georgia Institute of Technology*
Kurt Stirewalt, *Michigan State University*

Managing software maintenance projects is difficult. A manager typically must deal with a backlog of outstanding problems, a staff battling various concurrent fires, and a corporate profile in which failures have high visibility and success is greeted by a deafening silence. When a project includes programs written by a different group or possibly a different company, there's the added problem of trying to understand obscure foreign code. *Reverse engineering* is the process of comprehending software and producing a model of it at a high abstraction level, suitable for documentation, maintenance, or reengineering. Managers can use reverse engineering to better handle foreign code. In many projects, reverse-engineering technology has helped a maintenance team better understand a software system's structure and function. But from a manager's viewpoint, there are two painful problems:

- It's difficult or impossible to predict how much time reverse engineering will require.
- There are no standards to evaluate the quality of the reverse engineering that the maintenance staff performs.

Model-driven reverse engineering can overcome these difficulties. A *model* is a high-level representation of some aspect of a software system. Software engineers often use models to precisely specify systems before building them. In some cases, modeling tools can even generate part or all of the code without explicit, error-prone programming. MDRE uses these features of modeling technology but applies them differently to address the maintenance manager's problems. Our approach to MDRE uses formal specification and automatic code generation to reverse the reverse-engineering process. Models written in a formal specification language called SLANG describe both the application domain and the program being reverse engineered, and *inter-*

A model-driven reverse-engineering approach can help managers handle challenging software maintenance projects. This approach defines objective adequacy criteria for reverse engineering and provides a quality standard, enabling better effort prediction and quality evaluation and reducing

A compromise is to generate a program that can duplicate the original program's output without necessarily generating the same code.

pretations annotate the connections between the two. The ability to generate a similar version of a program gives managers a fixed target for reverse engineering. This, in turn, enables better effort prediction and quality evaluation, reducing development risk.

Adequate reverse engineering

The maintenance manager's uncertainty arises from a lack of understanding about when a reverse-engineering effort is adequate. The idea of adequacy comes from the world of software testing. An adequate test set is one that, when successfully executed, indicates that the software development's testing phase is complete.[1] Testers use various adequacy criteria, such as ensuring that tests exist to demonstrate the satisfactory exercise of all requirements or execution of all program statements. These criteria derive their benefit from being deterministic and measurable.

If adequacy criteria existed for reverse engineering, software engineers could collect experience reports and build databases of project statistics to help predict reverse-engineering time and effort. For such criteria to be useful, they must be objectively measurable. This implies there's an artifact that's the subject of the measurement. For testing, the artifact is the test suite. An adequate test suite provides confidence in the quality of the testing it directs. For MDRE, a maintenance manager measures adequacy using the high-level model produced by the software engineers undertaking the reverse engineering.

What might adequacy criteria for a reverse-engineering model comprise? Two characteristics seem essential: thoroughness and lucidity. *Thoroughness* is the extent to which the reverse-engineering effort covers the entire system under examination. *Lucidity* is the extent to which the reverse engineering sheds light on the system's purpose and how the code fulfills that purpose.

Reversing reverse engineering

How can models help measure a reverse-engineering effort's thoroughness and lucidity? The answer is to reverse the reverse-engineering process. In other words, we can use the result of reverse engineering to produce a second version of the original system. Thus, reverse engineering produces a high-level model of the system under study. If that model is expressed by a formal specification language supported by a code-generation tool, we can use the code generator to produce another version of the original system. If the generated version proves close enough to the original, the reverse-engineering effort was adequate.

However, what does "close enough" mean? Several possible definitions come to mind, corresponding to different degrees of adequacy. At one extreme would be a stub generator that produced a compilable version of the program, including stubs for all the subprograms and external data structures, but that produced no results when executed. Such tools currently exist, but they provide only superficial insight into a program's workings. At the other extreme would be a generator that reproduced the original program on a line-by-line basis. Although such a version might be desirable, it is currently unrealistic, given the state of specification technology and the allowable time available for reverse engineering.

A compromise is to generate a program that can duplicate the original program's output without necessarily generating the same code. That is, the close-enough version might not run as fast or use the same amount of memory, but it will compute the same values. This is the target we used in our MDRE approach.

Model-driven reverse engineering

What sort of models can support this reverse reverse engineering? MDRE uses two types, a program model and an application domain model, which roughly correspond to our target adequacy criteria for thoroughness and lucidity. In the following example, we describe both models using an algebraic specification language called SLANG, which is part of a tool called Specware,[2] developed by the Kestrel Institute. Specware includes a code generator capable of producing executable code from high-level models.

The *program model* provides a high-level rendering of the functions that the program computes. Thus, it provides a precise statement of the program-computed values but at a higher abstraction level than in the program source code. Algebraic specifications are popular precisely because they support programming at a high abstraction level, thereby reducing effort at the possible expense of execution speed. Because algebraic specifications are precise enough to serve as a basis for code generation, they enable measuring the thoroughness of reverse engineering.

An *application domain model* expresses domain concepts, their relationships, and their meanings independently of a program. An *application domain* is a set of related problems that have engendered software solutions.[3] An example is desktop publishing, for which there are several competing products. When a program solves an application domain problem, its users can expect a certain context, behavior, and terminology. Moreover, if the program's constructs can relate to corresponding domain model concepts, then it's possible to determine the roles these constructs play in achieving application domain goals. In MDRE, both the program model and the domain model are present, so we can make explicit connections between program constructs and the corresponding domain concepts. In this way, an application domain model is useful for assessing lucidity.

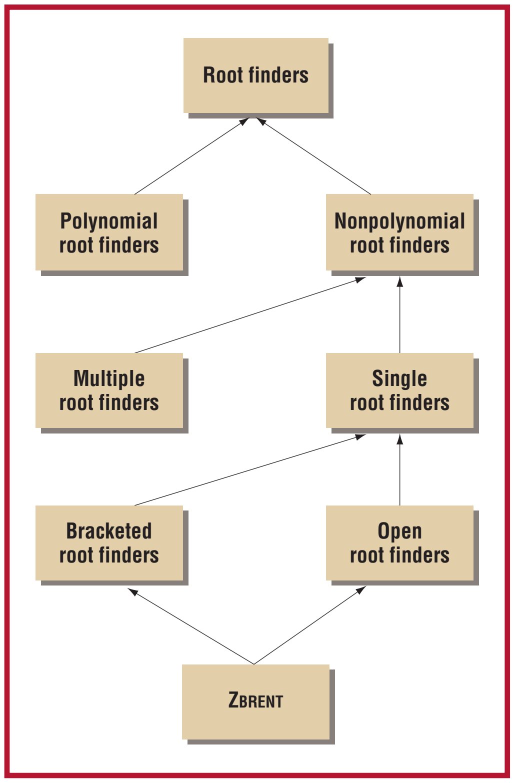


Figure 1. ZBRENT in the context of the root-finding domain. ZBRENT is a nonpolynomial, single-root finder that combines open and bracketed methods.

Example

To illustrate the issue of adequate models, we used reverse engineering on a numerical application called ZBRENT,[4] written in the C programming language. The application domain was numerical computation—specifically, finding the root of a real-valued function. We constructed a domain model by collecting material from textbooks on numerical analysis. Then we used SLANG to model both the domain and the program. Finally, we used the SLANG code generator in Specware to produce an executable version from our model of ZBRENT, which we compared with the original program on a set of test functions.

Root finding

Finding a nonlinear equation's root is a well-understood problem in numerical analysis.[5,6] There's a considerable collection of programs for finding roots. We can identify common characteristics from this collection and use them as expectations to guide the reverse-engineering process.

As Figure 1 shows, at the top level we can partition the root-finding application domain into polynomial and nonpolynomial algorithm families. For the latter, a further distinction exists between algorithms capable of finding multiple roots and those capable of handling only a single root. Our example concerns the latter class. In single-root finders, a final distinction exists between those guaranteed to converge (*bracketed* root finders) and those presumably more efficient ones that don't necessarily converge (*open* root finders).

Single-root, nonpolynomial root finders work as Figure 2 shows. The input includes a subprogram (f) to compute a functional value at a given real value (x), and an initial root estimate denoted by the endpoints of an interval within the set of real numbers. Functional eval-

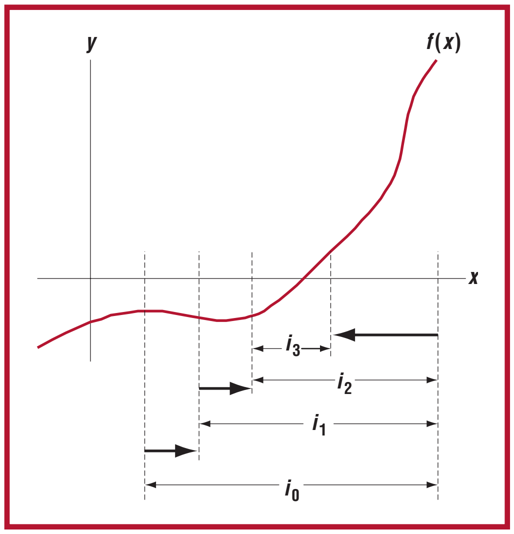


Figure 2. Iterative interval shrinkage.

```
(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.

uation is typically expensive, so root-finding algorithms try to reduce the number of calls to *f*.

Root finding normally involves selecting a trial point in the current interval, partitioning the interval into two pieces; determining the piece containing the root; creating a new, refined interval using the chosen piece; and iterating. In Figure 2, dashed lines indicate interval end points. Increasing subscripts in interval names denote the order of refinement. For example, interval i_3 refines interval i_2. For bracketed root-finding algorithms, the interval gets smaller with every iteration. Moreover, a stopping criterion determines whether there's been sufficient progress to warrant continuing the process.

Root-finding algorithms differ regarding the method for choosing a refined interval and the stopping criterion. Variations of the former include bisection (Bolzano's method), linear interpolation (Regula Falsi), inverse quadratic interpolation (Mueller's method), Aiken's delta-squared method, the Newton-Raphson method, and secant. Variations of algorithms that choose the stopping criterion include those that stop when the functional value is close enough to 0, the interval width is sufficiently narrow, or a fixed number of iterations have occurred. Of course, we can combine several methods to improve robustness or efficiency.

Zbrent

Zbrent is production software that's been in use for many years. It combines several of the variations just described, to improve efficiency and robustness. For our case study, we chose the Brent variation, from William Press and his colleagues,[7] for several reasons:

- It offers several stopping-criteria choices.
- It features three interval shrinkage methods: bisection, secant, and inverse quadratic interpolation.
- Victor Basili and Harlan Mills used a variant of Zbrent in an influential case study.[8] Thus, we can more closely compare our work with theirs.

Because of the number of variations, the code is complex and difficult to follow, making it a good candidate for reverse engineering.

Algebraic specification

Algebraic specifications consist of *sorts* (data types) and the *operations* that manipulate them. *Axioms* (sets of equations) define these operations. Each axiom equates the computed values of two different sequences of operations. We can think of the equations comprising an axiom as rewrite rules. Thus, we can replace occurrences of an equation's left-hand side with the right-hand side, possibly including parameter substitution. Specware automates the substitution to generate code that implements the operation in a programming language.

Figure 3 gives an example algebraic specification (in Slang code) of an interval that's used in building a root-finder domain model. This specification for `INTERVAL` (lines 1 to 20) describes a sort, `Interval` (line 3), that uses a previously defined sort, `Real`, to model *x*-axis values. An imported specification, `EXTENDED-REAL` (line 2), provides `Real`. A sort axiom (line 4) defines the structure of `Interval` as the Cartesian product of two `Real` sorts.

`INTERVAL` defines two operations: `mid-point` and `make-interval`. Operation definitions include a signature and one or more axioms. For example, the `mid-point` signature (line 6) indicates that it takes `Interval` as input and produces `Real` as output. The single axiom defining `mid-point` (lines 8 and 9) asserts that the output value produced, when its input is the `Interval` constructed from values `a` and `b`, equals the value produced from applying the `half` operation to the results of the `plus` operation for `a` and `b`. Similarly, lines 12 to 16 define the `make-interval` operation. Finally, lines 18 and 19 show that the `make-interval` operation provides a way to construct an `Interval`.

SLANG support for adequate models

Specifications in Specware are actual data values that high-level operators called *morphisms* can manipulate. SLANG provides three kinds of morphisms:

- *Import* includes one specification inside another.
- *Translate* renames a specification's sorts and operations.
- *Colimit* combines specifications in a structured way.

By writing simple specifications and then using morphisms to connect and compose them, developers can cleanly model complex systems.

We employed one other Specware feature, an *interpretation*, which Specware uses to formalize design refinements. Refinements relate abstract domain-model concepts to executable code. Operationally, an interpretation demonstrates how Specware implements sorts and operations in one model using sorts and operations in another model at a lower abstraction level. Interpretations let reverse engineers directly relate application domain concepts to program constructs. MDRE assesses lucidity by requiring interpretations to connect domains and implementations.

MDRE process

Our reverse engineering of ZBRENT included three steps. First, we constructed a domain model by reading descriptions in books and articles on root finding and articulating them in SLANG. The domain model provides expectations for concepts that root-finding programs might realize. Second, we constructed a program model by expressing the ZBRENT source code as a specification comprising a set of SLANG operation definitions. This step produces an abstract but comprehensive representation of the program without providing any insight into how the programming constructs relate to application domain concepts. Third, we defined SLANG interpretations using an iterative process to connect the program model operations to domain concepts. After making a set of connections, we executed the Specware code generator, producing an approximation to ZBRENT. If the generated program produced results identical to the original, the reverse engineering was thorough; if domain concepts connected to all the program constructs, the reverse engineering was lucid.

Introducing an interpretation often required refactoring the implementation model. We allowed only changes that maintained thoroughness—those for which Specware could generate a program that is *testing equivalent* (equivalent with respect to the outcome of testing) to ZBRENT's original code. We stopped this process when we could connect every implementation specification to the appropriate domain specification.

The root-finding domain model

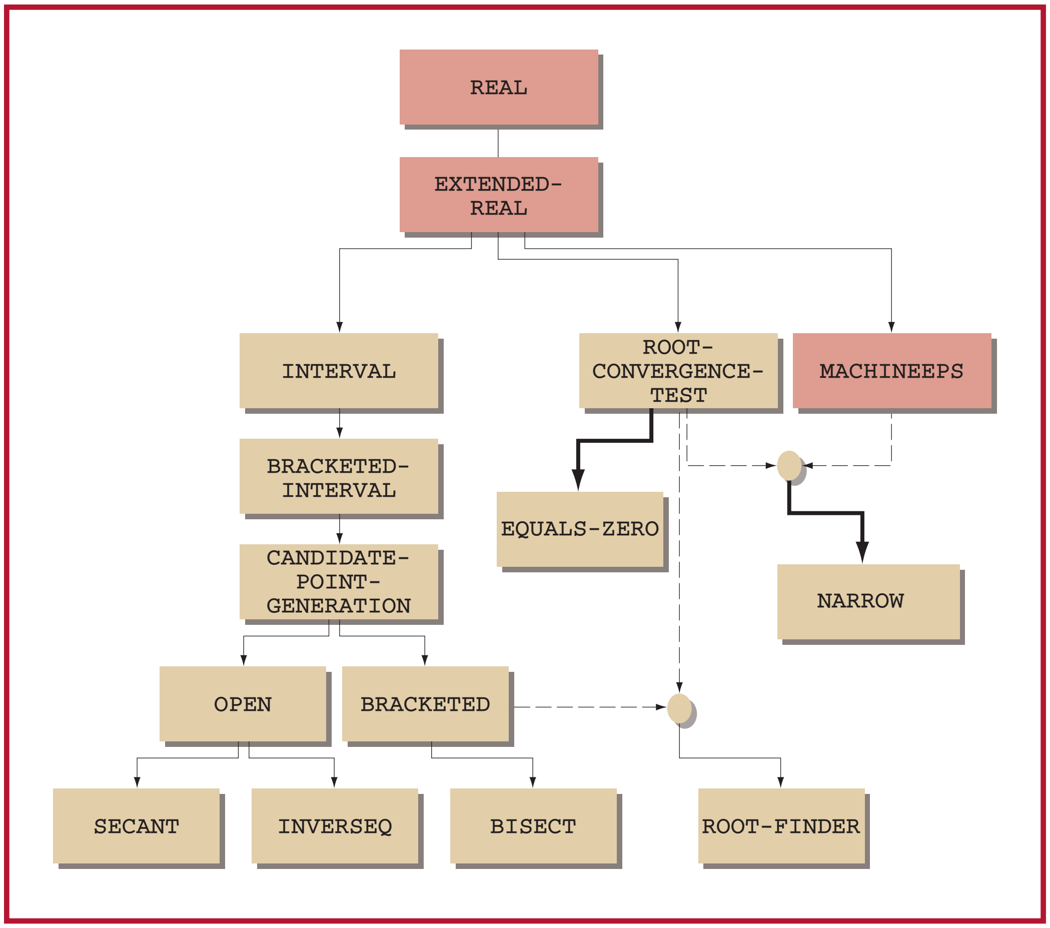
Figure 4 depicts the root-finding domain model as a set of related Specware specifications. Boxes denote specifications that represent different concepts and relationships in the domain. The darker boxes are not part of the root-finding domain but provide resources to it from other domains. For example, `REAL` is the specification for real numbers. The root-finding domain model proper includes 12 specifications, which we can organize into three groups: the iterative root-finding algorithm (`ROOT-FINDER`), termination condition checks (`ROOT-CONVERGENCE-TEST`, `NARROW`, and `EQUALS-ZERO`), and methods for interval shrinking—the remaining (lighter) boxes in Figure 4.

In addition to the 12 domain specifications, Figure 4 also illustrates several kinds of morphisms. An unadorned line denotes an import morphism, indicating the textual inclusion of one specification within another. This normally signifies a new specification that builds on an old one's features. For example, the `INTERVAL` specification must import from the `REAL` specification because the interval end points are real numbers. The bold lines correspond to translate morphisms, which rename one or more imported specification elements. In the root-finding domain model, renaming lets us write the `ROOT-FINDER` specification using an abstract convergence test. In the ZBRENT algorithm, the disjunction of the two concrete tests specified in the figure (`EQUALS-ZERO` and `NARROW`) refines this test.

Finally, in Figure 4, the dashed lines ending

By writing simple specifications and then using morphisms to connect and compose them, developers can cleanly model complex systems.

Figure 4. Root-finding domain model.



in a small circle denote the two colimit morphisms. A colimit is a shared union of the two source specifications. It's particularly valuable because it lets us separately model independent concepts and then explicitly combine them. For example, in Figure 4, a colimit produces the `NARROW` specification by combining the `ROOT-CONVERGENCE-TEST` and `MACHINEEPS` specifications. `ROOT-CONVERGENCE-TEST` defines properties of all mechanisms for terminating root-finding iterations; `MACHINEEPS` describes specific properties of floating-point numbers. `NARROW` then describes a test for termination when the current interval has become so small that no further progress is possible using available floating-point operations.

The ZBRENT program model

Besides the root-finding domain model, we needed a specification for the ZBRENT algorithm itself. The intent was to render the algorithm's details into SLANG so that we could use interpretations to relate them to the domain model. Figure 5 shows how this step works. In this high-level flow chart for ZBRENT, box labels express domain concepts. However, no such labels occur in the actual source code. Connections between source code constructs and domain concepts emerge only through significant iterative reverse engineering. The outer loop of the source code wraps the shrinking process and ensures termination by counting iterations. In this loop, it's possible to terminate successfully if either the root has been found (node labeled `root found`) or the interval itself has grown too narrow (`converged`). If termination is not warranted, ZBRENT checks to see whether interpolation is promising (`possible to interpolate`). In this case, the algorithm chooses either secant or inverse quadratic interpolation. If neither of these methods produces a bracketed value, then the algorithm uses bisection. Finally, ZBRENT updates the interval with the appropriately chosen subinterval.

The ZBRENT algorithm's SLANG specification implements the flow-chart boxes with axioms.

To construct this specification, reverse engineers must perform the following activities:

- Define operations corresponding to the various computations performed, nesting operation invocations where appropriate.
- Model conditional statements using built-in SLANG constructs.
- Use recursion to model iterative computations.
- Model assignments by passing the resulting state to subsequent operations.

During these steps, the software engineer must consider each program construct, thus increasing thoroughness. Moreover, the various operations and axioms produced become the targets of the interpretations devised during the third step of the reverse-engineering process.

Figure 6a contains a segment of the original program text for the box labeled "inverse quadratic interpolation" in Figure 5. Figure 6b contains the corresponding SLANG operation definition for line (3) of Figure 6a, the second computation of `q`. In Figure 6b, phrase `div(fa1,pToNZReal(fc1))` describes the earlier computation of `q` in line (1) of Figure 6a. Phrase `r(fb1, fc1)` describes the specification of program variable `r` in line (2) as an operation (also named `r`) applied to two parameters, `fb1` and `fc1`.

Interpretations

After constructing the ZBRENT algorithm's program model, the software engineer can define interpretations to indicate how to map domain concepts to program constructs. For the fragment shown in Figure 6a, the interpretation must map the domain specification of `INVERSEQ` to the set of operations that realize it in the program model.

The ZBRENT algorithm's SLANG model is thorough because Specware can automatically refine it into a program that is testing equivalent to the original source code. However, by itself, it sheds no light on how the algorithm finds a root. The SLANG interpretations serve this purpose. In particular, an interpretation indicates precisely how the program manifests an abstract domain concept. For example, an interpretation between the `EQUALS-ZERO` domain specification and the program model construct corresponding to the "Root found" box in Figure 5 gives a precise indication of the program construct's purpose, which is to use a root-convergence test to check program termination. To the extent that each component of the algorithm specification is formally related to an application domain concept through an interpretation, the maintenance manager judges that the domain model is a lucid representation of the program.

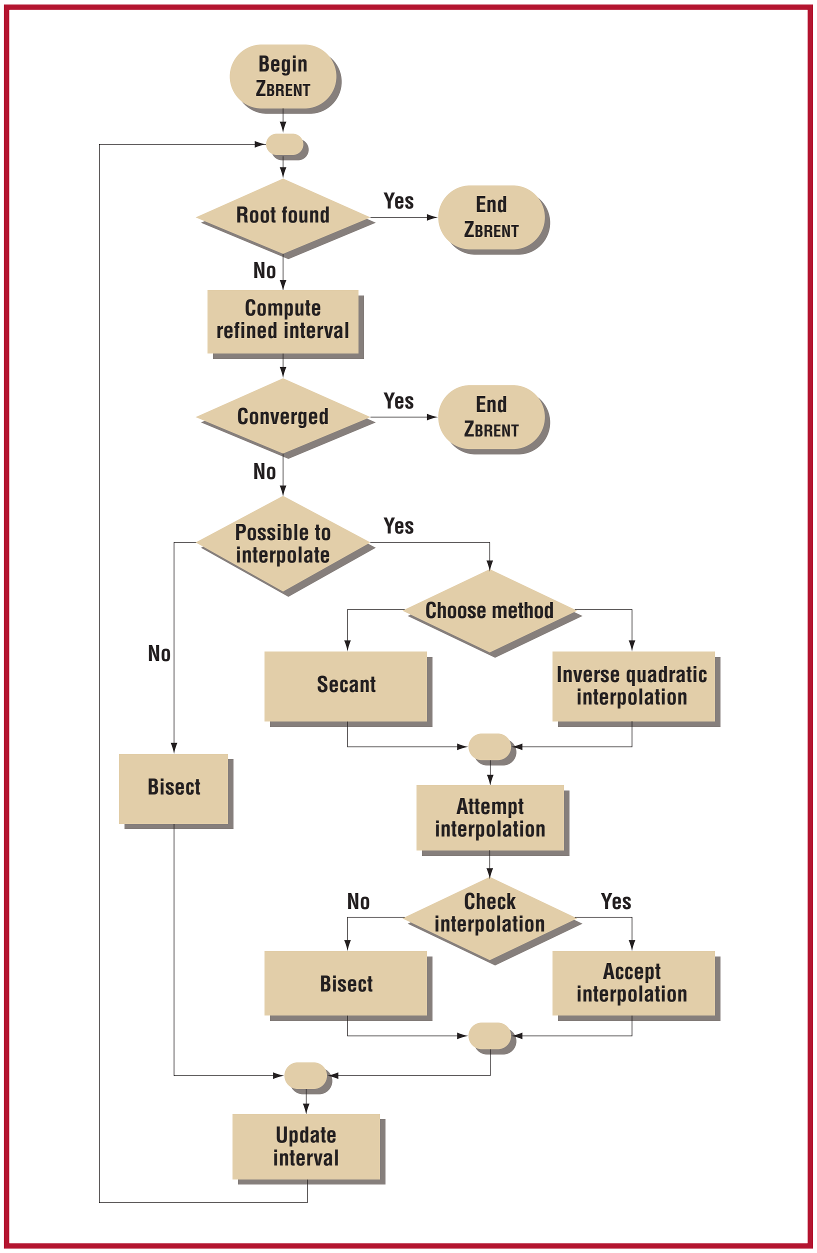


Figure 5. Abstract flow chart for ZBRENT.

Applying MDRE

A fixed standard for thoroughness and lucidity would let the maintenance manager better control the reverse-engineering process. There's an analogy with tools such as Co-

```
(1) q = fa / fc;
(2) r = fb / fc;
(3) q = (q - 1.0) * (r - 1.0) * (s - 1.0);
```
(a)

```
op q : Real, Real, Real, Real -> Real
   definition of q is
      axiom q(s, fa1, fb1, fc1) =
            times(minus(div(fa1,
                  pToNZReal(fc1)),one),
            times(minus(r(fb1, fc1),
                  one), minus(s, one)))
   end-definition
```
(b)

Figure 6. Inverse quadratic interpolation: (a) source code fragment; (b) SLANG specification.

como[9] and Slim[10] for estimating project schedules. These tools use a database of past experiences as a standard to evaluate current projects. Similar projects predict similar schedules. Likewise, adequacy standards for reverse-engineering efforts would allow maintenance managers to use experience data to predict the cost of such efforts.

An additional benefit of adequacy standards is as follows. Various reverse-engineering tools currently exist in the marketplace. But it's difficult to judge their benefits because there's no agreed-on standard to evaluate the quality of the representations they produce. An adequacy standard would allow direct comparisons—for example, indicating that one tool provides a more thorough description than another.

Relative adequacy

The adequacy criteria we presented are relative rather than absolute standards. Lucidity is relative to the abstraction level required for the reverse-engineering effort. Thoroughness is relative to the suite of tests used to determine equivalence; however, we can leverage what is known about adequate testing to provide an objective, deterministic standard.

Amortizing the cost of domain modeling

The reverse engineering of ZBRENT involved constructing a domain model for root finding, which required significant background research. Fortunately, this activity need occur only once regardless of how long the program lasts. Also, if there are other root-finding programs, they might be able to share the domain model. In other words, domain modeling has a value that goes beyond a single program's reverse engineering. We can amortize domain modeling's cost across subsequent maintenance activities for the same or related programs.

There are two prerequisites for using MDRE: a mature domain and a code-generation tool capable of compiling domain-specific application specifications. An example of a mature domain is scheduling—for instance, scheduling airline crews. Moreover, developers have used Specware to generate efficient scheduling algorithms from schedule constraints expressed in SLANG.[11]

The Unified Modeling Language[12] makes it possible to apply MDRE to a broader class of problems using alternative tool support. UML provides an industry-standard, semiformal design notation. Even though extra training and effort are necessary to use this notation, it's quite popular. UML provides various notations to use for modeling domains. Moreover, its graphical orientation, tool support, and early detection of certain error classes, along with the increasing popularity of the object-oriented methods it supports, suggest that it will provide some of the same advantages as formal methods. A recent addition to UML, the Object Constraint Language, complements UML's ability to describe structure with the ability to formally model program functionality. Together, UML and OCL facilitate more rigorous software development practices. Developers frequently use UML in designing information systems and e-commerce software. Applying MDRE in this context requires substituting a UML refinement tool, such as UML All Purpose Transformer,[13] for Specware, and a UML compatible language, such as OCL, for SLANG.

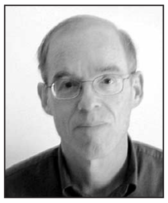
Acknowledgments

The National Science Foundation supported part of this work under grants CCR-9708913, EIA-0000433, and CCR-9984726. We also appreciate Terry Shikano's help in developing the example presented in this article. Thanks to the Kestrel Institute for providing Specware to us.

References

1. E.J. Weyuker, "Axiomatizing Software Test Data Adequacy," *IEEE Trans. Software Eng.*, Dec. 1986, pp. 1128–1138.
2. *Specware User Guide: Specware 2.0.3*, Kestrel Inst. and Kestrel Development Corp., 1998.
3. G. Arango and R. Prieto-Díaz, *Domain Analysis and Software Systems Modeling*, IEEE CS Press, 1991.
4. S. Rugaber, T. Shikano, and K. Stirewalt, "Adequate Reverse Engineering," *Proc. 16th IEEE Int'l Conf. Automated Software Eng.*, IEEE CS Press, 2001, pp. 232–244.
5. G. Dalhquist and Å. Björck, *Numerical Methods*, Prentice-Hall, 1974.
6. G.E. Forsythe, M.A. Malcolm, and C.B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1977.
7. W.H. Press et al., *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed., Cambridge University Press, 1992.
8. V.R. Basili and H.D. Mills, "Understanding and Documenting Programs," *IEEE Trans. Software Eng.*, May 1982, pp. 270–283.
9. B.W. Boehm et al., "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0," *Annals of Software Eng. 1*, Nov. 1995, pp. 57–94.
10. L.H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimation Problem," *IEEE Trans. Software Eng.*, July 1978, pp. 345–361.
11. D.R. Smith and S. Kambhampati, *Automated Synthesis of Planners and Schedulers*, project report, Kestrel Inst., 13 May 1997; www.kestrel.edu/HTML/projects/arpa-plan2/index.html.
12. Object Management Group, *OMG Unified Modeling Language Specification*, v. 1.4, OMG document 01-09-67, 2001; www.omg.org/cgi-bin/apps/doc?formal/01-09-67.pdf.
13. W.M. Ho et al., "UMLAUT: An Extendible UML Transformation Framework," *Proc. 14th IEEE Int'l Conf. Automated Software Eng.*, IEEE CS Press, 1999, pp. 275–278.

About the Authors



Spencer Rugaber is a senior research scientist in the College of Computing at the Georgia Institute of Technology. His research interests include software generation and reverse engineering. He received his PhD in computer science from Yale University. He is a member of the IEEE Computer Society and the ACM. Contact him at College of Computing, Georgia Tech., Atlanta, GA 30332-0280; spencer@cc.gatech.edu.

Kurt Stirewalt is an assistant professor in the Department of Computer Science and Engineering at Michigan State University. His research interests include the practical use of formal and semiformal graphical models in the design, verification, and maintenance of large software systems. He received his PhD in computer science from the Georgia Institute of Technology. He is a member of the IEEE Computer Society and ACM SIGSOFT. Contact him at the Dept. of Computer Science and Eng., Michigan State Univ., East Lansing, MI 48824; stire@cse.msu.edu.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.