# Model-Driven Reverse Engineering

Spencer Rugaber (spencer@cc.gatech.edu)
Georgia Institute of Technology
Kurt Stirewalt (stire@cps.msu.edu)
Michigan State University

Managing software maintenance projects is difficult. A manager typically has to deal with a backlog of outstanding problems, a staff battling numerous concurrent fires, and a corporate profile in which failures have high visibility and success is greeted by a deafening silence. And when a project being managed includes programs that were written by a different group or possibly a different company, there is the additional 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 level of abstraction, suitable for documentation, maintenance or reengineering. How can reverse engineering be used to help solve the problem of foreign code? Reverse engineering technology has, in fact, proven useful on many projects in helping a maintenance team obtain a better understanding of the structure and functioning of a software system. But, from a manager's point of view, there are two painful problems: it is difficult or impossible to predict how much time a reverse engineering effort will require, and there are no standards for how to evaluate the quality of the reverse engineering work the maintenance staff performs.

*Model-driven reverse engineering* (MDRE) is designed to overcome these difficulties. A *model* is a high-level representation of some aspect of a software system. One popular use of models by software engineers is to precisely specify systems before they are built. In some cases, modeling tools can even generate part or all of the code without having to resort to explicit and error-prone programming. MDRE uses these features of modeling technology but applies them differently to address the maintenance manager's problems.

## Adequate reverse engineering

The maintenance manager's uncertainty arises from the fact that we do not really understand when a reverse engineering effort is adequate. The idea of *adequacy* comes from the world of software testing where an adequate test set is one that, when successfully executed, indicates that the testing phase of development is complete [13]. Testers use various adequacy criteria, such as ensuring that tests exist to demonstrate that all requirements have been exercised or that all program statements have been executed. These criteria derive their benefit from being deterministic and measurable.

If adequacy criteria existed for reverse engineering, then software engineers could start collecting experience reports and building databases of project statistics to help predict reverse engineering time and effort.

For an adequacy criterion to be useful, it must be objectively measurable. This implies that there must be an artifact that is the subject of the measurement. In the case of testing, the artifact is the test suite. An adequate test suite is one that provides confidence in the quality of the testing it directs. In the case of MDRE, adequacy is measured 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 being examined. By *lucidity*, we mean the extent to which the reverse engineering sheds light on the purpose of the system and how that purpose is accomplished by the code.

## *Reverse* reverse engineering

How can models help with measuring the thoroughness and lucidity of a reverse engineering effort? The answer comes from the idea of *reversing* the reverse engineering process—taking the result of the reverse engineering and using it to produce a second version of the original system. In other words, reverse engineering produces a high-level model of the system being studied. If the model is expressed in a formal specification language and if the formal specification language is supported by a code generation tool, then the code generator can be used to produce another version of the original system. If the generated version proves close enough to the original, then we can have confidence that we have done an adequate job in our reverse engineering.

What does *close enough* mean? Several definitions 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 of the subprograms and external data structures, but that produced no results when executed. Such tools currently exist, but they provide only superficial insight into the workings of a program.

At the other extreme would be a generator that reproduced the original program on a line-by-line basis. While this version may be ultimately desirable, its seems unrealistic given the current state of specification technology and the allowable time available for reverse engineering.

A middle ground that addresses these concerns obtains if we can generate a program capable of duplicating the output of the original program on identical input without necessarily generating the same code. That is, the *close-enough* version may not run as fast or use the same amount of memory, but it does compute the same values. This is the target we have used in our model-driven reverse engineering efforts.

## Model-driven reverse engineering

What sort of models can support reverse reverse engineering? MDRE uses two sorts, a program model and an application-domain model, which roughly correspond to our target adequacy criteria for thoroughness and lucidity. In the example below, we describe both models using an algebraic specification language called SLANG. SLANG is part of a tool called Specware [7], developed by the Kestrel Institute, that includes a code generator capable of producing executable code from high-level models.

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

In contrast with a program model, an *application-domain model* expresses domain concepts, their relationships, and their meanings in a program-independent way. An *application domain* is a set of related problems that have engendered software solutions [1]. An example is desktop publishing for which numerous competing products have been developed. When a program is a member of an application domain, its users can expect a certain context, behavior, and terminology. Moreover, if the constructs in the program can be

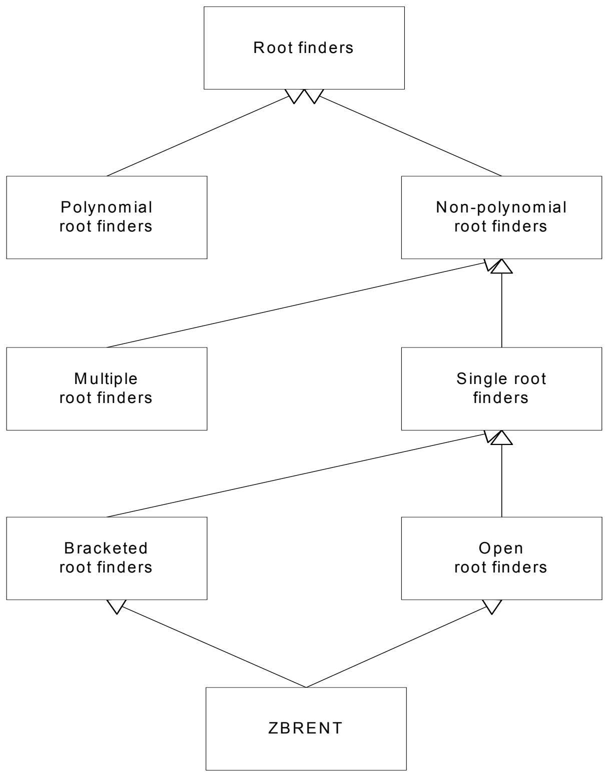

**Figure 1.** Lineage for ZBRENT

related to corresponding aspects of the domain model, then the roles these constructs play in achieving application domain goals can be determined. In MDRE, because both the program model and the domain model are present, explicit connections between program constructs and the corresponding domain concepts can be made. In this way, an application domain model enables lucidity to be assessed.

**Applicability of MDRE to other domains.** 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 such a domain is scheduling, for example, the scheduling of airline crews. The domain is mature and circumscribed. Moreover, Specware has been used to generate efficient scheduling algorithms from schedule constraints expressed in SLANG [12].

One recent technology, UML, suggest that MDRE could be applied to a broader class of problems making use of alternative tool support. UML (the Unified Modeling Language) [8], provides an industry standard, semi-formal design notation, which, although requiring extra training and effort to use, has proved popular. UML provides a variety of notations that can be used for modeling domains. Its graphical orientation, tool support, and early detection of certain classes of errors, together with the increasing popularity of the object-oriented methods it supports, suggest that it will provide some of the advantages claimed by proponents of formal methods. Moreover, a relatively recent addition to UML, OCL (the Object Constraint Language), complements UML's ability to describe structure with the ability to formally model program functionality. Together, these technologies provide developers a path to more rigorous software development practices. UML is frequently used in the design of information systems and e-commerce software. To apply MDRE in this context requires substituting a UML-refinement tool, such as UMLaut [6], for Specware and a UML compatible language, such as OCL, for SLANG.

## Example

To illustrate the issue of adequate models, we reverse engineered a numeric application called ZBRENT, written in the C programming language [11]. The application domain was numerical computation, specifically, finding the root of a real-valued function. A domain model was constructed by collecting material from textbooks on numerical analysis. Then SLANG was used to model both the domain and the program. We made use of the SLANG code generator in Specware to produce an executable version from our model of ZBRENT, which was then compared with the original program on a set of test functions.

**Root finding.** Finding the root of a nonlinear equation is a well-understood problem in numerical analysis [4][5]. That is, there is a

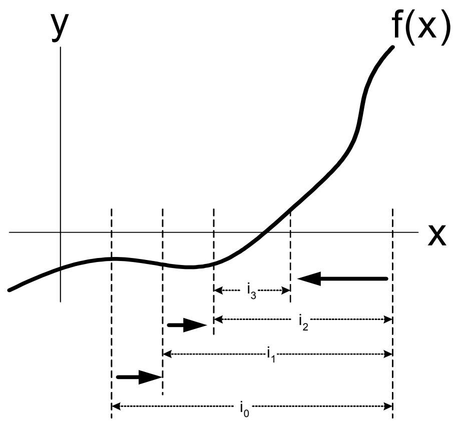

**Figure 2.** Iterative Interval Shrinkage

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 example, 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 a 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 f.

Root finding normally 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 $i_3$. Bracketed 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 kind can be categorized in two ways: first, by specifying the method in which a refined interval is chosen and, second, by the choice of stopping criterion. Variations of the former 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 latter 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 can be combined to improve robustness or efficiency.

**ZBRENT.** ZBRENT is production software that has been in use for many years. It combines several of the variations described above to improve efficiency and robustness. We chose the Brent variation, taken from [9], as the subject of our case study for several reasons.

- It features several choices of stopping criteria.
- It features three interval shrinkage methods: bisection, secant, and inverse quadratic interpolation.
- A variant of ZBRENT was used in an influential case study by Basili and Mills [2]. This allows us to more closely compare our work with theirs.

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

```
( 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

**Algebraic specification.** Algebraic specifications are composed using *sorts* (data types) and the *operations* that manipulate them. Operations are in turn defined via *axioms* (sets of equations). Each axiom indicates how the value computed by one sequence of operations equates to that computed by another.

The equations that comprise an axiom can also be thought of as rewrite rules; that is, occurrences of the left hand side of an equation can be replaced by the right hand side, possibly including parameter substitution. In Specware, the substitution can be automated in such a way that code can be generated that implements the operation in a programming language.

As an example of an algebraic specification, the SLANG code shown in Figure 3 presents 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 sort Real that models x-axis values. Real is provided by an imported specification called EXTENDED-REAL (line 2). The structure of an Interval is defined with a sort axiom (line 4) as being the Cartesian product of two Reals.

In INTERVAL, two operations are defined (mid-point and make-interval). Operation definitions consist of a signature and one or more axioms. For example, the signature for mid-point (line 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 applied to the results of applying the plus operation to a and b. 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 constructed.

**SLANG support for adequate models.** In Specware, specifications are actual data values that can be manipulated by high-level operators called *morphisms*. In particular, SLANG provides three kinds of morphisms: *import*, for including one specification inside 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 morphisms to operate on them, complex systems can be cleanly modeled.

We employed one other Specware feature called an *interpretation*, which Specware uses to formalize design refinements. Refinements relate abstract domain-model concepts to executable code. Operationally, an interpretation demonstrates how the sorts and operations in one model are implemented using sorts and operations in another model at a lower level of abstraction. Interpretations enable a reverse engineer to directly relate application-domain concepts to program constructs. MDRE assesses lucidity by requiring the use of interpretations to connect domain and implementation.

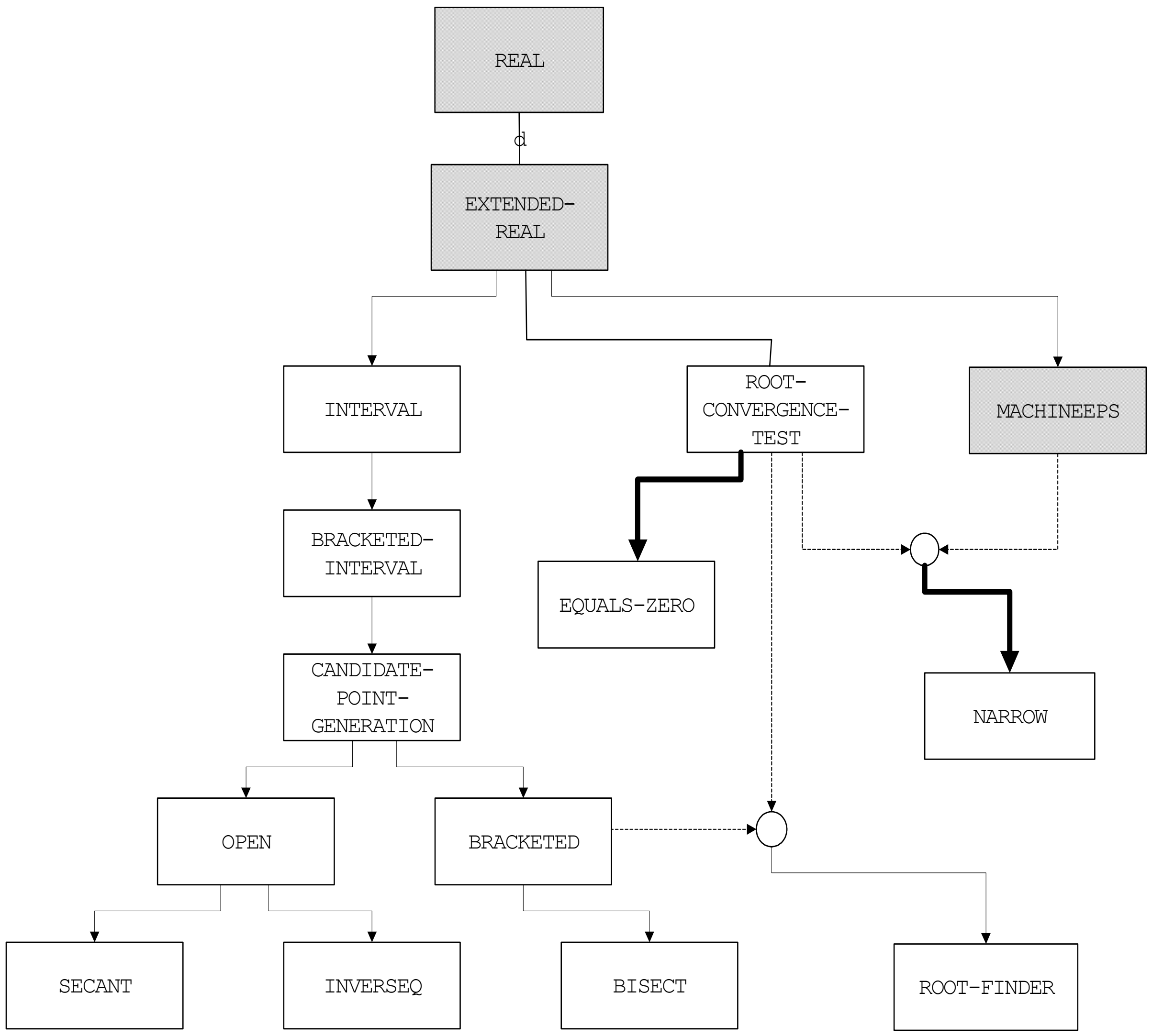

**Figure 4.** Root Finding Domain Model

**MDRE process.** The reverse engineering of ZBRENT consisted of three steps. In the first step, we constructed a domain model by reading descriptions in books and papers on root finding and articulating them in SLANG. The domain model provides a set of expectations for concepts that might be realized in root finding programs. In the second step, 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. During the third step, we engaged in an iterative process of defining SLANG interpretations to connect the program-model operations to domain concepts. After a set of connections are made, the Specware code generator is executed, producing an approximation to ZBRENT. If the generated program produces results identical to the original, then the reverse engineering is thorough; if all the program constructs have been connected with domain concepts, the reverse engineering is lucid.

Often, introducing an interpretation required refactoring the implementation model. We only allowed changes that maintained thoroughness; that is, ones for which Specware could generate a program that was testing equivalent to the original code for ZBRENT. We stopped this process when we were able to 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 in the figure denote specifications that represent different concepts and relationships in the domain. Filled 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 consists of twelve specifications, which can be organized 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 unfilled boxes).

In addition to the twelve domain specifications, Figure 4 also illustrates several kinds of morphisms. An unadorned line in the figure denotes an import morphism, indicating the textual inclusion of one specification within another. This normally signifies a new specification that builds on 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 the root-finding domain model, the renaming enables 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 (EQUALS-ZERO and NARROW).

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 particularly valuable because it allows independent concepts to be separately modeled and then explicitly combined. For example, in Figure 4, the ROOT-CONVERGENCE-TEST and MACHINEEPS specifications are combined using a colimit to produce the NARROW specification. 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 gotten so small that no further progress can be made using available floating-point operations.

**The ZBRENT program model.** In addition to the root-finding domain model, a specification must be given for the ZBRENT algorithm itself. The intent is to render the details of the algorithm into SLANG so that interpretations can then be used to relate them to the domain model. To illustrate how this step works, we make use of the flow-chart shown in Figure 5.

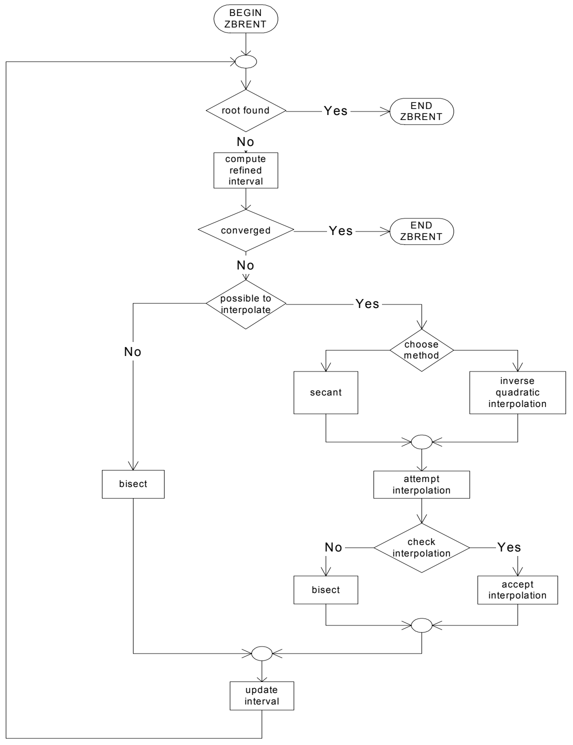

**Figure 5.** Abstract Flow Chart for ZBRENT

Figure 5 contains a high-level flow chart for ZBRENT. Although the box labels in the figure are expressed in terms of domain concepts, no such labels occur in the source code. Connections between source code constructs and domain concepts only emerge through significant iterative reverse engineering. The outer loop of the source code wraps the shrinking process and ensures termination by counting iterations. Within this loop, it is 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, a check is made to see whether interpolation is promising (possible to interpolate). In this case, either secant or inverse quadratic interpolation is chosen. If neither of these methods produces a bracketed value, then bisection is used. Finally, the interval is updated with the appropriately chosen subinterval.

The SLANG specification of the ZBRENT algorithm consists of a rendering of the flow chart boxes with axioms. In essence, this comprises the following activities:

- Define operations corresponding to the various computations performed, nesting operation invocations where appropriate.
- Make use of a built-in SLANG construct for rendering conditional statements.
- Use recursion to model iteration.
- Render assignments by passing the resulting state to subsequent operations.

During the course of performing these steps, a software engineer is forced to pay attention to each of the program constructs, thereby 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, the (emboldened) phrase div(fa1,pToNZReal(fc1)) describes the earlier computation of q on line (1) of Figure 6a; the (emboldened) phrase r(fb1,fc1) describes the specification of (the program variable) r on line (2) as an operation (also named) r applied to two parameters, fb1 and fc1.

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

**Figure 6a.** Source Code Fragment for Inverse Quadratic Interpolation Computation

```
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
```

**Figure 6b.** SLANG Specification for Inverse Quadratic Interpolation

**Interpretations.** Once the program model is constructed for the ZBRENT algorithm, interpretations can be defined to indicate how domain concepts are mapped to program constructs. For the fragment shown in Figure 6, an interpretation must be defined that maps the domain specification of INVERSEQ to the set of operations that realize it in the program model.

The SLANG model of the ZBRENT algorithm is thorough in the sense that a program equivalent to the original can be generated from it. However, by itself, it sheds no light on how the algorithm accomplishes the goal of finding a root. The SLANG interpretations are used for this purpose. In particular, an interpretation indicates precisely how an abstract domain concept is manifested in the program. For example, an interpretation between the domain specification EQUALS-ZERO and the program model construct corresponding to the box labeled root found gives a precise indication of the purpose of the program construct (program termination check). To the extent that each aspect of the algorithm specification is tied to an application-domain concept, the combined domain and program models are judged lucid.

## Applying model-driven reverse engineering

A fixed standard for thoroughness and lucidity would enable the maintenance manager to better control the reverse engineering process. An analogy exists with tools like COCOMO [3] and SLIM [10] that are used to estimate project schedules. These tools use a database of past experiences as a standard against which current projects can be measured. Similar projects predict similar schedules. Likewise, if adequacy standards for reverse engineering efforts existed, then experience data could be collected and used to predict reverse engineering effort.

An additional benefit accrues to adequacy standards. Currently, a variety of reverse engineering tools exist in the commercial marketplace. But their benefits are hard to judge because there is no agreed-upon standard for the quality of the representations they produce. An adequacy standard would allow direct comparisons to be made, indicating, for example, that one tool provides a more thorough description than another.

**Relative adequacy.** The adequacy criteria presented above were relative rather than absolute standards. For example, lucidity is relative to the level of abstraction required of the reverse engineering. Thoroughness is also relative to the suite of tests used to determine equivalence. Here, we leverage what is known about adequate testing to provide an objective and deterministic standard.

**Amortizing the cost of domain modeling.**

The process of reverse engineering ZBRENT included the construction of a domain model for root finding, requiring significant background research. Fortunately, this activity only has to be done once regardless of how long the program is maintained. And if other root finding programs are being maintained, it is likely that they can share the domain model. In other words, domain modeling has a value that goes beyond the reverse engineering of a single program—its cost can be amortized across subsequent maintenance activities for the same and related programs.

We have described a particular approach to Model-Driven Reverse Engineering. The approach uses a formal specification model and code generator to reverse the reverse engineering process. Models are used to describe both the application domain and the program being reverse engineered. Interpretations are used to annotate the connections between the two. Being able to generate a similar version of a program allows managers to have a fixed target for a reverse engineering effort. This, in turn, can enable better effort prediction and quality evaluation, thereby reducing development risk.

## Acknowledgement

Part of the work described here was supported by the National Science Foundation under grants CCR-9708913, EIA-0000433 and CCR-9984726. We also appreciate the contributions of Terry Shikano to the development of the example presented in this paper.

## 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] Barry W. Boehm, Bradford Clark, Ellis Horowitz, J. Christopher Westland, Raymond J. Madachy and Richard W. Selby. “Cost Models for Future Software Life Cycle Processes: COCOMO 2.0." *Annals of Software Engineering,* 1, pp. 57-94, 1995.

[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] Wai Ming Ho, J.-M. Jezequel, A. Le Guennec and F. Pennaneac'h. “UMLAUT: An Extendible UML Transformation Framework.” *14th IEEE International Conference on Automated Software Engineering, 1999,* October 1999, pp. 275-278.

[7] Kestrel Institute. *Specware User Guide,* Version 2.0.3. March 1998.

[8] Object Management Group. “Unified Modeling Language, Version 1.4.” OMG Document Number 01-09-67, http://www.omg.org/cgi-bin/apps/doc?formal/01-09-67.pdf.

[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] L. H. Putnam. “A General Empirical Solution to the Macro Software Sizing and Estimation Problem.” *IEEE Transaction on Software Engineering,* Volume SE-4, Number 4, July 1978.

[11] Spencer Rugaber, Terry Shikano, and Kurt Stirewalt. “Adequate Reverse Engineering.” *Proceedings of the Conference on Automated Software Engineering,* San Diego California, 2001.

[12] Douglas R. Smith and Subbarao Kambhampati. “Automated Synthesis of Planners and Schedulers.” http://www.kestrel.edu/HTML/projects/arpa-plan2/index.html.

[13] Elaine J. Weyuker. “Axiomatizing Software Test Data Adequacy.” *IEEE Transactions on Software Engineering,* SE-12(12): 1128-1138, December 1986.