Teaching the Essential Role of Mathematical Modeling in Understanding and Reasoning About Objects

Murali Sitaraman
Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV 26506

Bruce W. Weide, Timothy J. Long
Computer and Information Science
The Ohio State University
Columbus, OH 43210

Wayne Heym
Mathematical Sciences
Otterbein College
Westerville, OH 43081

murali@cs.wvu.edu
{long,weide}@cis.ohio-state.edu
wheym@otterbein.edu

Technical report OSU-CISRC-9/97-TR43
September 1997

Abstract

Recognizing the advantages of object-based software, CS instructors have increasingly emphasized design and reuse of "objects" in CS1/CS2. But unless students also learn how to understand objects and operations abstractly, and how to reason about the behavior of object-based programs, they can't possibly write correct non-trivial software. We describe how we present formal yet simple mathematical models of objects and operations and how we use them to teach reasoning about object-based program behavior. The techniques we teach are both suitable for introductory courses and scalable to industrial-strength software.

1 Introduction

There is a tremendous gap between the kinds of real world information we write programs to process, and the bits of information that computer hardware ultimately is able to process. So generally we don't program in terms of bits. Using the built-in types provided by our programming languages and higher-level user-defined types dramatically simplifies software development.

But *why* exactly do types simplify software development? Here are two traditional explanations:

- Using types allows the compiler to catch many common programming errors. In this view, types are important because they help eliminate an entire class of run-time errors.
- Reusing previously-defined types means less new code needs to be written, because the representation data structures and operations for those types have already been coded by someone else. In this view, types are important because they save coding time.

Both these observations are valid. But they miss the crux of the issue: The essential role of types is that they *simplify human understanding and reasoning about the behavior of software systems*. In this paper we explain why types — or, more specifically, the mathematical models attached to types — are necessary in any practical discipline for building industrial-strength software systems. We also explain how we teach this to our students in CS1/CS2, and illustrate with a CS2-level example. A companion paper [2] provides context by describing our overall approach to CS1/CS2 and the place of mathematical modeling in the big picture we give students in this sequence.

2 The Reasoning Problem

One of the many skills required to create a working program is the ability to solve the ***reasoning problem***: the ability to reason soundly about the behavior of a sequence of statements without actually executing it on a computer [5]. The argument for this is rather straightforward. Suppose we could not reason about what a statement does, by using only some mental or paper-and-pencil process; that we had to ask the computer to execute it to see what happens. Then how would we know what statements to ask the computer to execute in the first place? So, an effective solution to the reasoning problem is a crucial piece of any practical software development paradigm.

To see how objects and types support a successful attack on the reasoning problem, consider a common built-in type such as *Integer*. How do we, as programmers, understand what code involving *Integer* objects (variables) does? Do we view *Integer* objects as, e.g., 32-bit entities, and operators "+ "and "–" as macros that stand for microprograms of control sequences that manipulate 32-bit entities? This is fine for the hardware designer who is *implementing* arithmetic circuits to manipulate bits, but it is at best ridiculously inconvenient for the software engineer who is a *client* of that hardware.

For programming purposes we much prefer to envision *Integer* objects as though they were mathematical integers, possibly bounded by implementation-specific constants. From the choice of names, we also imagine *Integer* operators such as "+" and "–" as though they performed additions and subtractions of mathematical integers. In other words, we don't think about *Integer* objects in terms of internal representations but in terms of implementation-neutral (i.e., "abstract") ***mathematical models***. The type *Integer* gives us a hook on which to hang an appropriate mathematical model, i.e., mathematical integers.

The reasoning problem needs to be addressed for every type, including user-defined types that arise routinely in object-based and object-oriented software systems. Consider this piece of code involving *List* objects:

```
procedure Reverse (alters s: List) is
    temp: Item
  begin
    if Length (s) > 0 then
      Remove (s, temp)
      Reverse (s)
      Insert (s, temp)
    end if
  end Reverse
```

Assuming we understand that the intended objective of *Reverse* is to "reverse" a *List*, how do we reason about whether this body actually accomplishes that? We need to know exactly what a *List* is and exactly what each of the operations *Length*, *Remove*, and *Insert* does. Mathematical modeling seem like an obvious approach.

But is it really so obvious? To see how such a question might be answered in CS1/CS2, we examined several textbook descriptions of "List" and "Insert" and found a wide range of explanations, from the nearly-content-free to the cryptic to the implementation-dependent. Here (without attribution) are a few of the explanations of the effect of an *Insert* operation for a *List* object:

- a new item is inserted into a list;
- post condition: the list = the list + the item;
- insert adds the item to the beginning of the pointer "pre" (accompanied by a figure showing a typical configuration of a linked list representation with a "pre" pointer, among others)

Evidently, if we really want to know what *Insert* does, we need to understand a specific linked list representation and to examine and understand the code for the body of the *Insert* operation. Then we need to apply the same sledgehammer to understand what *Remove* does. Finally, we might "manually execute" the *Reverse* code multiple times on multiple inputs, at which point we might make an educated guess about whether *Reverse* works as intended.

Using and reasoning about objects of a user-defined type such as *List*, without an abstract explanation of the type and operations using explicit mathematical models, is akin to reasoning about *Integer* objects as though they were sequences of bits. How should a user-defined type such as *List* and its operations be explained, given that a basic objective of software engineering is to be able to reason about and understand the software we write? The rest of this paper sketches what we teach as the answer to this fundamental question — one which must be addressed no matter which programming language or paradigm is used.

3 An Example: Reversing a List

Figure 1 shows the specification of a *List* type in a dialect of the RESOLVE notation [3]. *List_Template* is a generic **concept** (specification template) which is parameterized by the type of items to be contained in lists. To provide abstract mathematical explanations of the operations, an object of type *List* **is modeled by** an *ordered pair of mathematical strings* of items.[1] Conceptualizing a *List* object as a pair of strings makes it easy to explain insertion and removal from the "middle". A sample value of a *List* of *Integer*s object, for example, is the ordered pair (<3,4,5>,<4,1>). Insertions and removals can be explained as taking place between the two strings, i.e., at the right end of the left string or at the left end of the right string.

The declaration of type *List* introduces the mathematical model and says that an object of type *List* initially (i.e., upon declaration) is "empty": both its left and right strings are empty strings. Each operation is specified by a **requires** clause (precondition), which is an obligation for the caller; and an **ensures** clause (postcondition), which is a guarantee from a correct implementation. In the postcondition of *Insert*, for example, *#s* and *#x* denote the incoming values of *s* and *x*, respectively, and *s* and x denote the outgoing values. *Insert* has no requirement, and it ensures that the incoming value of *x* is concatenated onto the left end of the right string of the incoming value of *s*; the left string is not affected. Notice that the postcondition describes how the operation **alters** the value of *s*, but the return value of *x* (which has the mode **consumes**) remains unspecified.

RESOLVE specifications use a combination of standard mathematical models such as integers, sets, functions, and relations, in addition to tuples and strings. The explicit introduction of mathematical models allows use of standard notations associated with those models in explaining the operations. Our experience is that this notation — which is precise and formal — is nonetheless fairly easy to understand for CS1/CS2 students because they have seen most of it before, in high school and earlier. When using programming languages such as Ada, C++, or Java for lab assignments, we embed such RESOLVE specifications as "formal comments" in order to define mathematical models for user-defined types and to give precise behavioral specifications of operations.

```
concept List_Template (type Item)

    type List is modeled by (
        left: string of Item
        right: string of Item
      )
     exemplar s
     initialization ensures
       |s.left| = 0 and |s.right| = 0

   operation Insert (
        alters  s: List
        consumes x: Item
      )
     ensures  s.left = #s.left and
              s.right = <#x> * #s.right

   operation Remove (
        alters  s: List
        produces x: Item
      )
     requires |s.right| > 0
     ensures  s.left = #s.left and
              #s.right = <x> * s.right

   operation Advance (
        alters  s: List
      )
     requires |s.right| > 0
     ensures  s.left * s.right =
                #s.left * #s.right and
              |s.left| = |#s.left| + 1

   operation Reset (
        alters  s: List
      )
     ensures   |s.left| = 0 and
            s.right = #s.left * #s.right

   operation Advance_To_End (
        alters  s: List
      )
     ensures   |s.right| = 0 and
              s.left = #s.left * #s.right

   operation Left_Length (
        preserves s: List
      ): Integer
     ensures  Left_Length = |s.left|

   operation Right_Length (
        preserves s: List
      ): Integer
     ensures  Right_Length = |s.right|

end List_Template
```

Figure 1 — RESOLVE specification of *List*s

---

1 A string is similar to, but simpler than, a "sequence" because it does not explicitly include the notion of a position. The operator "*" denotes string concatenation; "<*x*>" denotes the string consisting of the single item *x*; and "|*s*|" denotes the length of string *s*. sR denotes the reversal of a string.

State	Facts	Obligations
0	`s = (<>, <3, 4, 6, 2>)` **and** `temp = 0`	
if `Right_Length (s) > 0` **then**		
1	`s = (<>, <3, 4, 6, 2>)` **and** `temp = 0`	`|s.right| > 0?`
`Remove (s, temp)`		
2	`s = (<>, <4, 6, 2>)` **and** `temp = 3`	`|s.left| = 0?` **and** `|s.right| < |<3, 4, 6, 2>|?`
`Reverse (s)`		
3	`s = (<2, 6, 4>, <>)` **and** `temp = 3`	
`Insert (s, temp)`		
4	`s = (<2, 6, 4>, <3>)` **and** `temp = 0`	
end if		
5	`s = (<2, 6, 4>, <3>)` **and** `temp = 0`	`s = (<2, 6, 4, 3>, <>)?`

Figure 2 — A tracing table for *Reverse*

We leave to the reader the task of understanding the other *List_Template* operations. *List_Template* is typical of the specifications students see in our CS1/CS2 sequences. Other types covered in these courses include queues and stacks, bags, partial maps, etc. Students also see several application-specific types specified in a similar way [2].

4 Reasoning About *Reverse*

Now we return to the problem of reasoning about the *Reverse* operation. Shown below is one possible formal specification of the *Reverse* operation, i.e., this is what we intend to mean by "reversing" a *List* object:

```
operation Reverse (
    alters s: List
  )
  requires |s.left| = 0
  ensures  s.left = reverse (#s.right) and
           |s.right| = 0
```

Let's reconsider the question raised earlier (where *Length* has been replaced in the code with *Right_Length* to be consistent with Figure 1): Is the following implementation correct for the above specification of *Reverse*?

```
procedure Reverse (alters s: List) is
   temp: Item
  begin
    if Right_Length (s) > 0 then
      Remove (s, temp)
      Reverse (s)
      Insert (s, temp)
    end if
  end Reverse
```

The only new notation here is **reverse**, a built-in mathematical function in the specification notation. Its intuitive meaning is that, if *s* is a string (e.g., <1,2,3>), then **reverse** (*s*) is the string whose elements are the same as those in *s* but in the opposite order (e.g., <3,2,1>).

Reasoning about correctness can be done with varying degrees of confidence through ***testing*** (automated execution on sample inputs), ***tracing*** (mental and/or manual execution on sample inputs), or formal ***symbolic reasoning*** — but all of these must be based on mathematical modeling of *List*s. In our CS1/CS2 courses, students achieve familiarity with testing and mastery of tracing, but usually are only introduced to formal symbolic reasoning. Although testing is clearly important, here we illustrate only the last two approaches to show the power of mathematical modeling for reasoning about program behavior.

The reasoning method we teach is ***natural reasoning***, a technique proposed by Heym [1], who also proved its soundness and relative completeness. The method is called *natural* reasoning, like natural deduction in mathematics, because it is an operationally-based approach that is intuitively appealing to CS1/CS2 students and experienced software engineers alike. Natural reasoning starts with program states, i.e., resting points between statements at which the values of objects might be observed. Three questions about every state are important for reasoning about the behavior of the code:

- Under what condition can the program get into this state?
- Under that condition, what do we know about the values of the objects in this state?
- Under that condition, what must be true of the values of the objects in this state in order that the next statement should work properly, i.e., in order that the program can successfully move to the next state?

The columns in Figures 2 and 3 that follow **State** contain answers to these questions for that state. Column **Path condition** (Figure 3 only) gives the condition under which execution reaches the state. Column **Facts** enumerates assumptions (such as the ensures clauses of called operations) that can be made in the state. Column **Obligations** lists assertions (such as the requires clauses of called operations) that need to be proved in the state in order for execution to proceed smoothly to the next state.

<table>
<tr><th></th><th>State</th><th>Path condition</th><th>Facts</th><th>Obligations</th></tr>
<tr><td></td><td>0</td><td></td><td>$|s_0.left| = 0$ and
is_initial $(temp_0)$</td><td></td></tr>
<tr><td colspan="5">if Right_Length (s) > 0 then</td></tr>
<tr><td></td><td>1</td><td>$|s_0.right| > 0$</td><td>$s_1 = s_0$ and
$temp_1 = temp_0$</td><td>$|s_1.right| > 0$</td></tr>
<tr><td colspan="5">Remove (s, temp)</td></tr>
<tr><td></td><td>2</td><td>$|s_0.right| > 0$</td><td>$s_2.left = s_1.left$ and
$s_1.right = \langle temp_2\rangle * s_2.right$</td><td>$|s_2.left| = 0$ and
|s2.right| < |s0.right|</td></tr>
<tr><td colspan="5">Reverse (s)</td></tr>
<tr><td></td><td>3</td><td>$|s_0.right| > 0$</td><td>$s_3.left = (s_2.right)R$ and
$|s_3.right| = 0$ and
$temp_3 = temp_2$</td><td></td></tr>
<tr><td colspan="5">Insert (s, temp)</td></tr>
<tr><td></td><td>4</td><td>$|s_0.right| > 0$</td><td>$s_4.left = s_3.left$ and
$s_4.right = \langle temp_3\rangle * s_3.right$</td><td></td></tr>
<tr><td colspan="5">end if</td></tr>
<tr><td></td><td>5a</td><td>$|s_0.right| = 0$</td><td>$s_5 = s_0$ and
$temp_5 = temp_0$</td><td rowspan="2">$s_5.left =$ reverse $(s_0.right)$ and
$|s_5.right| = 0$</td></tr>
<tr><td></td><td>5b</td><td>$|s_0.right| > 0$</td><td>$s_5 = s_4$ and
$temp_5 = temp_4$</td></tr>
</table>

Figure 3 — A symbolic reasoning table for *Reverse*

Figure 2 shows a tracing table with s_0 = (<>,<3,4,6,2>), which satisfies the requires clause of *Reverse*. At the end of the code, the ensures clause of *Reverse* must be satisfied, i.e., we must have s_5 = (<2, 6, 4, 3>,<>).

In this tracing table execution passes through all states because the **if** condition holds for the input. In state 1, for example, there is an obligation $|s.right| > 0$? because of the requires clause of *Remove*. In state 2, the ensures clause of *Remove* leads to the stated facts. Similarly, the obligations in state 2 and the facts in state 3 arise, respectively, from the requires and ensures clauses of the (recursive) call to *Reverse*. Natural reasoning uses a built-in induction argument here so recursion is nothing special, except that before a recursive call there is an additional obligation to show termination: the recursive operation's *progress metric* has decreased. Details of the rest of this table are similar. For the code to be correct — on this input only, of course — every obligation must be shown to be true using facts in states before and including the state of the obligation.

Figure 3 illustrates symbolic reasoning, which is a bit more complex but substantially more powerful. Using s_0 to denote the value of s in state 0, and so on, facts and obligations are written based on specifications; this is similar to tracing. Additionally, path conditions (such as $|s_0.right| > 0$ for states 1-4) that describe when states are reached. Once all the verification conditions are filled in, the reasoning problem is solved by combining them appropriately to show that all the obligations are satisfied.

The soundness of natural reasoning depends upon using only the following in the proof of an obligation for state *k*:

- (path condition for state *i*) **implies** (facts for state *i*), for every *i* satisfying $0 \leq i \leq k$, and
- the path condition for state *k*.

So, to prove the obligation in state 1 we may assume:

$|s_0.left| = 0$ **and**
is_initial $(temp_0)$ **and**
$|s_0.right| > 0$ **implies**
$(s_1 = s_0$ **and** $temp_1 = temp_0)$ **and**
$|s_0.right| > 0$

The proof of this obligation (like most) is quite simple. Knowing that $|s_0.right| > 0$ we conclude $s_1 = s_0$ and, therefore, $s_1.right = s_0.right$. Since $|s_0.right| > 0$ we conclude $|s_1.right| > 0$, i.e., the obligation in state 1 which was to be proved. It is our experience so far that students usually can handle such proofs. So we discuss many other reasoning examples, including ones with loops and *loop invariants*, in our CS1/CS2 courses [4, 5].

This process, of course, can get tedious, but tool-based generation of facts and obligations is straightforward and could be developed for professional and classroom use. However, for a student being introduced to reasoning about program behavior, a few exercises such as the one here help to drive home some important points. One of the key points is that it would be (almost literally) impossible to reason soundly about the correctness of even simple code like the body of *Reverse* without a mathematical model of *List* and specifications for *List* operations in terms of that mathematical model.

Is *Reverse* correct? The tracing table clearly shows a counterexample to any claim of correctness. If the code were correct, however, tracing could not show this whereas formal symbolic reasoning could. Fixing the program is left as an exercise, as we might leave it to our students.

5 Conclusion

We teach the ideas presented in this paper in CS1/CS2 at WVU [4] and at OSU [5]. Our approach to teaching a discipline of software engineering is rather different from traditional CS1/CS2 content [2]. But the importance of types and mathematical modeling for understanding and reasoning about program behavior transcends the differences among CS1/CS2 approaches.

The sample problem illustrates the importance of mathematical modeling in reasoning about software. Without precise descriptions based on mathematical models, the benefits of object-oriented development and reuse are offset because clients who use existing components will be unable to understand those components well enough to reason soundly about non-trivial programs which use them. The implications of this for software quality are ominous.

6 Acknowledgment

We gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants CCR-9311702, DUE-9555062, and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, and from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, or the U.S. Department of Defense.

7 References

[1] Heym, W.D. *Computer Program Verification: Improvements for Human Reasoning*. Ph.D. diss., Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, 1995.

[2] Long, T.J, et al. *Providing Intellectual Focus to CS1/CS2*. Tech. report OSU-CISRC-9/97-TR42, Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, Sept. 1997.

[3] Sitaraman, M., and Weide, B.W., eds. Component-based software using RESOLVE. *ACM Software Eng. Notes 19*, 4 (1994), 21-67.

[4] Sitaraman, M. *An Introduction to Software Engineering Using Properly Conceptualized Objects*. WVU Publications, 1997.

[5] Weide, B.W. *Software Component Engineering*. OSU Reprographics, Columbus, 1997.