Reasoning about Software-Component Behavior

Murali Sitaraman[1], Steven Atkinson[1], Gregory Kulczycki[1], Bruce W. Weide[2], Timothy J. Long[2], Paolo Bucci[2], Wayne Heym[2], Scott Pike[2], and Joseph E. Hollingsworth[3]

[1] Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV 26506 USA
{murali,atkinson,gregwk}@csee.wvu.edu
[2] Computer and Information Science
The Ohio State University
Columbus, OH 43210 USA
{weide,long,bucci,heym,pike}@cis.ohio-state.edu
[3] Computer Science
Indiana University Southeast
New Albany, IN 47150 USA
jholly@ius.indiana.edu

Abstract. The correctness of a component-based software system depends on the component client's ability to reason about the behavior of the components that comprise the system, both in isolation and as composed. The soundness of such reasoning is dubious given the current state of the practice. Soundness is especially troublesome for component technologies where source code for some components is inherently unavailable to the client. Fortunately, there is a simple, understandable, teachable, practical, and provably sound and relatively complete reasoning system for component-based software systems that addresses the reasoning problem.

Keywords: component-based software, reasoning, software component, software reuse, specification, verification.

1 Introduction

Both the object-oriented literature and common sense suggest that component-based software development, and the resulting software reuse, should improve programmer productivity and software quality because:

- less new code must be written to produce the same results, and
- off-the-shelf components should be "well-seasoned" and therefore more reliable than code written from scratch.

Both these observations are basically valid. But they are not the main reason why component-based software has the potential to dramatically improve

W. B. Frakes (Ed.): ICSR-6, LNCS 1844, pp. 266-283, 2000.

software engineering practice. The key feature of well-designed software components is that they—or more specifically, the mathematical models used to explain them—can help you understand, and reason soundly about, the execution-time **behavior** of component-based software systems. Don Knuth emphasizes the importance of such reasoning in an interview in *Byte* magazine [8]:

> [People in the object computing realm] haven't yet built a reliable way to reason about these programs, that is, we still lack the mathematical proofs to ensure a program will work. With object oriented programs, we have much less of an understanding of how we would ever prove that they don't have bugs. This is a huge gap. If people can understand OOP, they ought to be able to prove that the programs are correct.

How can this problem be addressed, especially when some components in the program are not available in source form? In this paper we describe how to use mathematical modeling to explain and to reason about software-component behavior, i.e., the computational states reached during execution. We also demonstrate why you *must* use appropriate mathematical models if you expect to be able to reason about the composite behavior of software systems built from such components.

2 The Reasoning Problem

Any robust software-development paradigm must provide an answer to the **reasoning problem** [12, 17]; namely, how can you reason soundly about the behavior of a statement without actually executing it on a computer? The argument for this claim is straightforward. Suppose you could not reason abstractly about what a statement does, that you had to run it on a computer to see what happens. Then how would you choose a statement to ask the computer to execute? Trial-and-error is a surprisingly common approach for newcomers to computing, but it cannot work for software professionals because clearly there are just too many possible statements to try them all. You must be able to do *some* reasoning just to prune the options. A practical solution to the reasoning problem must be effective and reliable, not mere guesswork—even if you never try to "prove" anything about your programs.

Consider a common built-in programming type such as *Integer*. How, for example, do you reason about the effect of code involving objects (variables[1]) of type *Integer?* A hardware engineer might view the value of an *Integer* object as a boolean vector, and the high-level-language operators "+" and "–" as macros that stand for hardware control sequences which manipulate boolean vectors.

[1] We use the word "object" throughout this paper to emphasize that the techniques illustrated apply to object-oriented programs involving inheritance, etc., as well as to the "object-based" program fragments that constitute this paper's examples. For instance, [17] shows how the approach works with component-based C++ software; see also http://www.cis.ohio-state.edu/~weide/sce/now.

"Boolean vector" is an example of a **mathematical model** for the value of an *Integer* object, i.e., something that defines a mental image for the object's value and provides a machine-processable notation that supports formal reasoning about that object's behavior.

The boolean vector model for programming type *Integer* works well for the hardware designer who is implementing arithmetic circuits. But it is at best unnecessarily complex for the software engineer who is a client of that hardware. For a software engineering task, you normally view the value of an *Integer* object according to a more appropriate mathematical model: a mathematical integer. You also picture *Integer* operators such as "+" and "–" as performing additions and subtractions of mathematical integers. You don't think about *Integer* objects in terms of internal representations, but in terms of their representation-neutral (i.e., "abstract") mathematical models.

3 Reasoning with Software Components

Component-based software development aggravates the reasoning problem because it significantly widens the semantic gap between the kinds of real-world information you can write programs to process, and the bits that computer hardware ultimately is able to process. Appropriate mathematical models have long since been adopted for the built-in types provided by programming languages. But in component-based software development you use not only these built-in types—which are one or two levels removed from the hardware—but also much higher-level types defined by off-the-shelf software components with powerful operations whose exact behavior can be complex, and even mysterious if it is not very carefully described. What are appropriate mathematical models for these types?

The burgeoning popularity of component technologies, from the early Booch components [2] through such distributed object technology contenders [10] as CORBA, DCOM, and Jini, makes it imperative that reasoning difficulties with component-based software be dealt with before they lead to a software disaster. Fortunately, software components present an opportunity along with the reasoning challenge. Every programming type gives you something to "wrap" with an appropriate mathematical model. In fact, researchers have already used this idea to tie formal mathematical models to some popular-technology components [9]. The models involved are more complex than simple mathematical integers. But they are far less complex than the underlying bits used in computer representations and the code that transforms them, which must remain the last resort for understanding program behavior.

Mathematical modeling also provides guidance when trying to identify and design new domain-specific software components. Textbooks on the subject usually stop short of detailed component designs. They assume that the domain-specific concepts identified by analysis, if named appropriately, will be intrinsically understandable to domain experts through intuitive or metaphorical models (e.g., "a stack is like a stack of cafeteria trays"). But in complex domains

where system correctness is very important—such as air-traffic control—the precise behavioral details of software components must be so well-understood that specification by wishful naming and content-free explanations such as "a stack is like a stack" cannot suffice.

Moreover, the software objects in a system often do not correspond one-to-one with actual physical objects, making it impossible to explain the behavior of some software objects by appealing to physical analogies. Implementations of complex domain-specific components usually are layered over other complex components, making it practically impossible to understand their behavior by sifting through their implementation code. Finally, in many component technologies *no source code is available* for some or all components.

4 An Example: "List"

We might have used a software component from a domain such as air-traffic control as an example of selecting and using appropriate mathematical models. But the point is clear (perhaps even clearer) when the component in question deals with something most software engineers seem to know and "understand" very well. So consider this piece of code that uses *List* objects, where *List* is a programming type defined by an off-the-shelf software component:

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

Assuming you understand informally that the intended behavior of *Reverse* is to "reverse" a *List* object, how do you reason soundly about whether this body actually accomplishes that? You need to know exactly what a *List* object is, exactly what each of the operations *Length, Remove,* and *Insert* does, and exactly what *Reverse* is supposed to do. Mathematical modeling seems like an obvious approach.

But is this answer really so obvious? To see how such a question might be answered in traditional documentation for clients of a "List" component, we examined several descriptions of off-the-shelf components involving "List" and "Insert". We found a wide range of explanations ranging from the content-free to the cryptic to the implementation-dependent to the nearly acceptable (i.e., the best we could find). Here, quoted directly but without attribution, are a few of the explanations we found for the behavior of an *Insert* operation for a *List:*

- a new item is inserted into a list

- postcondition: 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]
- put v at ith position... ensures insertion_done: i_th(i) = v [accompanied by a separate definition of another programming operation called "i_th": item at ith position]

Evidently, if you want to know *exactly* what *Insert* does and does not do, you need to understand a specific linked list representation and the code for the body of the *Insert* operation. Then you need to apply the same sledgehammer to understand what *Remove* does. Finally, you can "manually execute" the *Reverse* code on multiple inputs, at which point you might make an educated guess about whether *Reverse* works as intended. Without an explicit mathematical model that abstractly specifies the state of a *List* object and the behavior of *List* operations, reasoning about *List* objects is reduced to speculation and guesswork.

How should objects and their operations be explained, given that a basic objective of software engineering is to be able to reason about and understand the software? The next section illustrates an answer to this fundamental question using the *List* example. The issue at hand is one that you must address no matter which programming language or paradigm you use. But it is especially important for component-based software development, where source code for the components used often is not available to the client programmer.

5 Explaining the Values and Behavior of Lists

To arrive at an appropriate mathematical model that explains the behavior of a *List* object, we start by considering exactly what values (states) and state changes we are trying to model. Figure 1 shows a common singly linked list data structure consisting of a sequence of nodes chained together by *next* pointers. New nodes can be added to or removed from the sequence just after the node referenced by *cur_pos*. Other operations allow the sequence of *data* items to be traversed by following *next* pointers. Evidently, in Figure 1, a traversal has already visited the consecutive nodes containing items 3, 4, and 5, and has yet to visit the remaining nodes containing items 1 and 4.

What is the essence of the information captured in this data structure, independent of its representation? We claim that it is simply the string[2] of items already visited, namely $\langle 3, 4, 5 \rangle$, and the string of items yet to be visited, namely $\langle 1, 4 \rangle$. That is, you can view the value of a *List* object as an ordered pair of mathematical strings of items.

As the *Integer*-as-boolean-vector example suggests, mathematical modeling does not by itself guarantee understandable specifications or ease of reasoning.

[2] A string is technically simpler than a "sequence" because it is finite and does not explicitly involve the notion of a position. But thinking of a string as a sequence will not lead you astray.

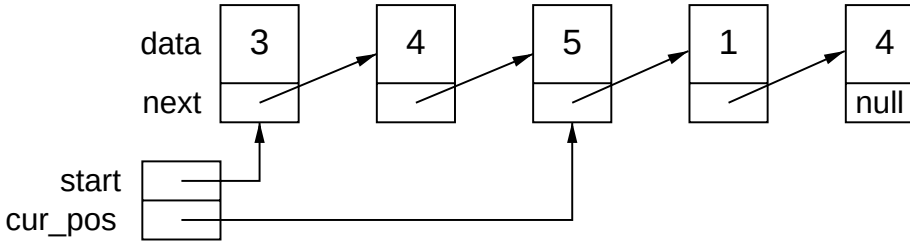

Fig. 1. A typical singly linked list representation

Choosing a *good* mathematical model is a crucial but sometimes difficult task. For example, you might choose to think of the value of a *List* object as a single string of items (e.g., $\langle 3, 4, 5, 1, 4 \rangle$) along with an integer current position (e.g., 3); as a function from integer positions to items, along with a current position; or even as a complex mathematical structure that captures the links and nodes of the above representation. Selection of a good mathematical model depends heavily on the operations to be specified, the choice of which should be guided by considerations of observability, controllability, and performance-influenced pragmatism [4, 16]. The pair-of-strings model suggested above leads (in our opinion) to the most understandable specification of the concept and makes it easy to reason about programs that use *List* objects, as we will see.

Figure 2 shows the specification of a *List* component in a dialect of the RESOLVE language [13]. *List_Template* is a generic **concept** (specification template) which is parameterized by the programming type of items in the lists. As just stated, each *List* object **is modeled by** an ordered pair of mathematical strings of items. The operator "*" denotes string concatenation; "$\langle x \rangle$" denotes the string consisting of a single item x; and "$|s|$" denotes the length of string s.

Conceptualizing a *List* object as a pair of strings makes it easy to explain the behavior of operations that insert or remove from the "middle". A sample value of a *List_Of_Integers* object, for example, is the ordered pair $(\langle 3, 4, 5 \rangle, \langle 1, 4 \rangle)$. Insertions and removals can be explained as taking place between the two strings, i.e., either at the right end of the left string or at the left end of the right string.

The declaration of the programming type *List* introduces the mathematical model and says that a *List* object 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 precondition, 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 **updates** the value of s, but the return value of parameter x (which has the mode **clears**)

```
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 ( updates s: List, clears x: Item )
    ensures s.left = #s.left and
      s.right = <#x> * #s.right

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

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

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

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

  operation Left_Length ( restores s: List )
      returns length: Integer
    ensures length = |s.left|

  operation Right_Length ( restores s: List )
      returns length: Integer
    ensures length = |s.right|

end List_Template
```

Fig. 2. RESOLVE specification of a *List* component

remains otherwise unspecified; **clears** means it gets an initial value for its type. For example, an *Integer* object has an initial value of 0.

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 the use of standard notations associated with those models in explaining the operations. Our experience is that this notation, while precise and formal, is nonetheless fairly easy to learn, even for beginning computer science students.

We leave to the reader the task of understanding the other *List_Template* operations. *List_Template* is just an example chosen to illustrate the features of explicit mathematical modeling as a specification approach. Other off-the-shelf RESOLVE components include general-purpose ones defining queues, stacks, bags, partial maps, sorting machines, solvers for graph optimization problems, etc.; and more complex domain-specific components.

6 Reasoning About Reverse

Shown below is one possible formal specification of *Reverse,* i.e., this is what we intend to mean by "reversing" a *List* object:

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

The only new notation here is **reverse**, a built-in mathematical function in the specification notation. Formally, its inductive definition is:

```
  reverse (empty_string) = empty_string
  reverse (a * <x>) = <x> * reverse (a)
```

Informally, its meaning is that, if s is a string (e.g., $\langle 1, 2, 3 \rangle$), then **reverse** (s) is the string whose items are those in s but in the opposite order (e.g., $\langle 3, 2, 1 \rangle$).

Let's reconsider the reasoning question raised earlier (where *Length* has been replaced in the code with *Right_Length* to match exactly the component interface defined in Figure 2). Is the following implementation correct for the above specification of *Reverse?*

```
procedure Reverse ( updates s: List )
    decreasing |s.right|
  begin
    variable temp: Item
    if Right_Length (s) > 0 then
      Remove (s, temp)
      Reverse (s)
      Insert (s, temp)
    end if
  end Reverse
```

You can reason about the correctness of this code with varying degrees of confidence through **testing** (computer execution on sample inputs), **tracing** (human execution on sample inputs), and/or **formal symbolic reasoning** (proof of correctness). But all of these must be based on mathematical modeling of *Lists.*

Table 1. A tracing table for *Reverse*

State	Facts
0	s = (<>, <3, 4, 6, 2>) **and** temp = 0
if Right_Length (s) > 0 **then**	
1	s = (<>, <3, 4, 6, 2>) **and** temp = 0
Remove (s, temp)	
2	s = (<>, <4, 6, 2>) **and** temp = 3
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

Although testing is clearly important, here we illustrate only the latter two approaches to show the power of mathematical modeling for *human* reasoning about program behavior.

Tracing. Tracing is sometimes part of code reviews, walkthroughs, and formal technical reviews [5]. It is helpful to use a conventional form when conducting a trace. Table 1 shows a **tracing table** for *Reverse* where the incoming value of the *List_Of_Integers s* is $(\langle\,\rangle, \langle 3, 4, 6, 2\rangle)$. The **Facts** column of this table records the values of objects in the corresponding state of the program listed in the **State** column. States are the "resting points" between statements at which values of objects might be observed.

There are two states in Table 1 where the recording of facts calls for some explanation. The facts at state 2 are based on the postcondition of the *Remove* operation. However, you can assume the postcondition of *Remove* only if the precondition of *Remove* is satisfied before the call, i.e., in state 1. In this case, object values at state 1 can be seen by inspection to satisfy the precondition of *Remove,* so appealing to the postcondition of *Remove* to characterize state 2 represents valid reasoning. Also, the facts at state 3 use the postcondition of

Reverse. Assuming the postcondition of *Reverse* when tracing *Reverse* would represent circular, invalid reasoning without first verifying that the recursion is "making progress". In this case, progress is evident because the length of *s.right,* at state 2, is less than the length of *s.right* at state 0. Again you can see this by inspection. The justification for appealing to the postcondition of *Reverse* in state 3 is, then, mathematical induction. (Note also that the precondition of *Reverse* holds at state 2.)

Details of the remaining entries of the table are straightforward. Examination of the facts at state 5 reveals whether this implementation of *Reverse* is correct for the specific input value $s = (\langle\,\rangle, \langle 3, 4, 6, 2\rangle)$. You should be able to see from this trace and the specification that it is *not* correct.

Formal Symbolic Reasoning. This is a powerful generalization of tracing where the names of objects stand for arbitrary values of the mathematical models of their types, not for specific values. For example, instead of tracing *Reverse* using the specific values $\#s.left = \langle\,\rangle$ and $\#s.right = \langle 3, 4, 6, 2\rangle$, you simply let $\#s.left$ and $\#s.right$ denote some arbitrary incoming value of s.

Our approach to symbolic reasoning is called **natural reasoning**, a verification technique proposed by Heym [7], who also proved conditions for its soundness and relative completeness. The general idea is called *natural* reasoning, like natural deduction in mathematics, because it is an operationally-based approach that is intuitively appealing to computer science students and experienced software engineers alike. It lets you formally represent the informal reasoning used by the author of the code, effectively encoding why he/she thinks the code "works".

Heym compared his natural reasoning system with classical program verification techniques based on Hoare logic, and with two earlier proposals for similar reasoning methods (which lacked soundness proofs and, as it turned out, were actually unsound). We do not digress here to discuss this related theoretical work; see [7] for details. Some other features of natural reasoning are:

- Programs with loops are handled through the use of traditional loop invariants or loop specifications, which are not illustrated in the present example.
- The techniques used in our *List Reverse* example generalize to cover reasoning about the correctness of data representations (e.g., to decide whether a proposed *List_Template* implementation is correct), also not illustrated here.
- The soundness of natural reasoning depends on the absence of aliasing in the client code. In RESOLVE, we eliminate aliasing by using the swapping paradigm [6] as the basis for component design, implementation, and use. The consequences of this decision are illustrated in the detailed design of *List_Template*, most notably in the way *Insert* works (note in Figure 2 the parameter mode for x).

Natural reasoning about code correctness can be viewed as a two-step process:

1. Record local information about the code in a **symbolic reasoning table**, a generalization of a tracing table.
2. Establish the code's correctness by combining the recorded information into, and then proving, the code's **verification conditions**.

Step 1 is a symbol-processing activity no more complex than compiling. It can be done automatically. Consider an operation *Foo* that has two parameters and whose body consists of a sequence of statements (Figure 3). You first examine *stmt-1* and record assertions that describe the relationship which results from it, involving the values of x and y in state 0 (call these x_0 and y_0) and in state 1 (x_1 and y_1). You similarly record the relationship which results from executing *stmt-2,* i.e., involving x_1, y_1, x_2, and y_2; and so on. You can do this for the statements in any order because these relationships are local, involving consecutive states of the program.

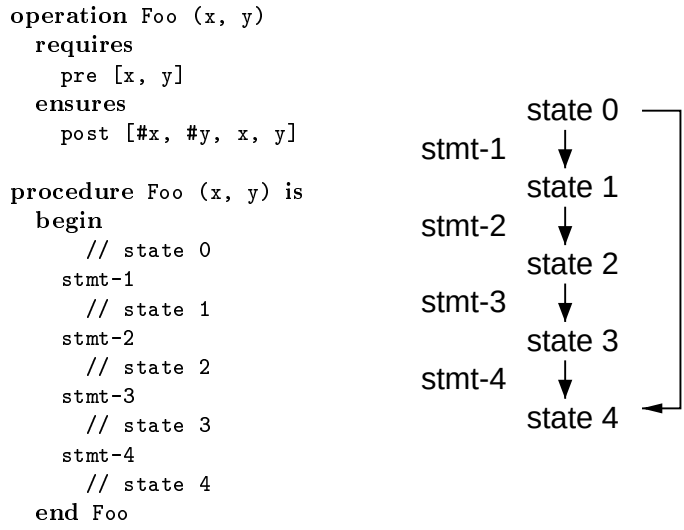

Fig. 3. Relationships in symbolic reasoning

In addition to those arising from the procedure body statements, step 1 produces two special assertions. One is a **fact** (an assertion to be assumed in step 2 of natural reasoning): the precondition of *Foo* holds in state 0, i.e., $pre[x_0, y_0]$. Another is an **obligation** (an assertion to be proved in step 2): the postcondition of *Foo* holds in state 4 with respect to state 0, i.e., $post[x_0, y_0, x_4, y_4]$. Intuitively, this says that if you view the effect of the operation from the client program, as control appears to jump directly from state 0 to state 4, the net effect of the individual statements in the body is consistent with the specification.

Step 2 of natural reasoning involves combining the assertions recorded in step 1 to show that all the obligations can be proved from the available facts. This task is generally an intellectually challenging activity in which computer-assisted theorem proving helps, but, given the current state-of-the-art, it is far from entirely automatic.

The assertions recorded in step 1 arise from three questions about every state:

– Under what condition can the program get into this state?

- If the program gets into this state, what do we know about the values of the objects?
- If the program gets into this state, what must be true of the values of the objects in order that the program can successfully move to the next state?

Table 2 shows a completed symbolic reasoning table for *Reverse*. The columns to the right of the **State** column contain the answers to the above questions for a given state. Column **Path Conditions** records the condition under which execution reaches that state. Column **Facts** records assumptions (generally the postconditions of called operations) that can be made in that state. Column **Obligations** records assertions (generally the preconditions of called operations) that need to be true in that state in order for execution to proceed smoothly to the next state.

Table 2. A symbolic reasoning table for *Reverse*

State	Path Conditions	Facts	Obligations
0		|s_0.left| = 0 **and** **is_initial** (temp$_0$)	
if Right_Length (s) > 0 **then**			
1	|s_0.right| > 0	s_1 = s_0 **and** temp$_1$ = temp$_0$	|s_1.right| > 0
Remove (s, temp)			
2	|s_0.right| > 0	s_2.left = s_1.left **and** s_1.right = <temp$_2$> * s_2.right	|s_2.left| = 0 **and** |s_2.right| < |s_0.right|
Reverse (s)			
3	|s_0.right| > 0	s_3.left = **reverse** (s_2.right) **and** |s_3.right| = 0 **and** temp$_3$ = temp$_2$	
Insert (s, temp)			
4	|s_0.right| > 0	s_4.left = s_3.left **and** s_4.right = <temp$_3$> * s_3.right **and** **is_initial** (temp$_4$)	
end if			
5a	|s_0.right| = 0	s_5 = s_0 **and** temp$_5$ = temp$_0$	s_5.left = **reverse** (s_0.right) **and** |s_5.right| = 0
5b	|s_0.right| > 0	s_5 = s_4 **and** temp$_5$ = temp$_4$	

In Table 2, $s_i.left$ and $s_i.right$ are the symbolic denotations of values for object s in state i; similarly for object *temp*. The facts at state 0 are obtained

by substituting the symbolic value of object s at state 0, namely s_0, into the precondition of *Reverse,* and by recording initial values for all local objects. The obligation at state 5 is obtained by substituting the symbolic values of s at state 0 and at state 5 into the postcondition of *Reverse.* This is the goal obligation: when it is also proved, the correctness of *Reverse* is established.

Notice how the path condition $|s_0.right| > 0$ for states 1–4 records when these states are reached. Facts recorded for states 1–5 are based on the postconditions of operations and on the flow of control for an **if** statement. Obligations arise in state 1, because of the precondition of *Remove*, and in state 2, because of the precondition of *Reverse* and because *Reverse* is being called recursively. Natural reasoning includes a built-in induction argument here so recursion is nothing special, except that before a recursive call there is an obligation to show termination: the recursive operation's progress metric has decreased, in this case, $|s_2.right| < |s_0.right|$. A progress metric, like a loop invariant, is a claim that must be supplied by the programmer of a recursive body; hence, the **decreasing** clause in the body of *Reverse.* A proof obligation just before any recursive call is that the claim holds, i.e., that execution is making progress along this metric.

Once all these assertions are recorded, you solve the reasoning problem by composing them appropriately to form the verification conditions and then showing that each of these conditions is satisfied. There is one verification condition for each obligation, of the form:

assumptions **implies** obligation

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

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

So, in order to discharge the proof of the obligation in state 1 of Table 2, i.e., $|s_1.right| > 0$, you may assume:

(true implies ($|\mathtt{s}_0\mathtt{.left}| = 0$ **and is_initial (**$\mathtt{temp}_0$**)) and**
($|\mathtt{s}_0\mathtt{.right}| > 0$ **implies (**$\mathtt{s}_1 = \mathtt{s}_0$ **and** $\mathtt{temp}_1 = \mathtt{temp}_0$**)) and**
$|\mathtt{s}_0\mathtt{.right}| > 0$

The first two conjuncts are the assumptions of the first form for states 0 and 1, respectively, and the third is the assumption of the second form for state 1.

The proof of the obligation in state 1 is easy for humans who have had a bit of practice with such things. Assuming that $|s_0.right| > 0$, you conclude from the second line that $s_1 = s_0$ and, therefore, $s_1.right = s_0.right$. Then since $|s_0.right| > 0$ you conclude by substitution $|s_1.right| > 0$, i.e., the assertion to be proved. In a similar manner, you can easily prove the obligation at state 2.

Is *Reverse* correct? Table 1 shows a counterexample to a claim of correctness; indeed the obligation at state 5 cannot be proved from the allowable assumptions. If the code were correct, however, tracing could not show this whereas symbolic

reasoning could. Fixing the program is left as an exercise for the reader, as we would leave it for our students.

7 Experience

We routinely introduce mathematical modeling and the important role of specifications in reasoning, using the RESOLVE notation, in first-year CS course sequences at The Ohio State University (OSU) and West Virginia University (WVU). We have conducted formal attitudinal and content-based surveys as well as essay-style evaluations to assess the impact of teaching these principles. A detailed summary of the results to date is beyond the scope of this paper. But the evaluations with a sample size over 100 allow us to reach at least the following interim conclusions:

- Most students can learn to understand mathematical modeling as the basis for explanations of object behavior. This is illustrated by their ability to select reusable components and to act as clients of components, without *any* knowledge of those components' implementations. It is confirmed by their performance on exam questions asking them to write operation bodies and test plans, given only formal specifications (often involving quantifiers).
- After the course sequence, a statistically significant number of students have changed certain attitudes about programming. They tend to believe at the end of the sequence (but not before starting it) that natural language descriptions are inadequate descriptions of software components, and that it is possible to show that a software component works correctly without running it on a computer.

A prototype implementation of the tracing and natural reasoning systems described in this article is part of the Software Composition Workbench tool being developed by the Reusable Software Research Groups at WVU and OSU.[3] The tool generates symbolic reasoning tables automatically. It then uses the PVS theorem prover [11] to discharge the verification conditions. The prover typically requires human intervention and advice in this process.

8 Formalism: Necessity and Scalability

This section addresses two questions that we often get concerning the necessity for and scalability of formal mathematical modeling and formal natural reasoning.

The first of these involves necessity: Given that non-trivial component-based software systems are routinely built and deployed without using mathematical models for describing component behavior, is such precision and care really

[3] For examples of symbolic reasoning tables generated by this system, see http://www.csee.wvu.edu/~resolve/scw.

necessary? There is little doubt that component designers, implementers, and clients generally agree on certain unwritten conventions to go along with informal natural-language component documentation. The result is that *usually* components are used where they will not "break". In other words, most component-based software is written under wishful usage assumptions that more or less hold—but not always. An example of such a situation involves the inadvertent introduction of aliasing through repeated arguments, as noted in [3]:

> The most obvious forms of aliasing are similar to those found in any system involving references, pointers, or any other link-like construct. For example, in the matrix multiplication routine:
>
> ```
> op matMul(lft: Matrix, rgt: Matrix, result: Matrix);
> ```
>
> It may be the case that two or more of lft, rgt, and result are actually connected to the same matrix object. If the usual matrix multiplication algorithm is applied, and result is bound to the same matrix as either of the sources, the procedure will produce incorrect results.

The reason for using explicit mathematical modeling, and for adhering to a formalized reasoning method whose conditions for soundness have been established, is to limit or eliminate this dependence on wishful usage assumptions. Where the consequences of software failures are significant enough, the economics justify the added expense of more careful modeling and more careful reasoning.

This brings us to the second question, regarding that expense and the resulting trade-off, which is often phrased in terms of scalability: Even granting that it is (sometimes) necessary, can formal mathematical modeling and formal natural reasoning scale up to handle practical, large, and complex components and systems?

The first answer is that the techniques of using mathematical modeling for precise description of component behavior, and of using natural reasoning to predict the behavior of software built from such components, are independent of the complexity of the components and the size of the systems built from them. In fact, we (and many others) have developed formal specifications for quite a few components that are far more complex than *Lists*; e.g., see [1]. Such examples, however, clearly have required a serious amount of modeling and specification effort. Admittedly, the fact that such work appears in research papers (which is appropriate, given the current state of the art) does little to raise confidence that formal mathematical modeling and formal natural reasoning are practical approaches usable by "real programmers."

A second answer can be obtained by revisiting the premise behind the first question: Somehow, the same people whose ability to reason formally about software system behavior is being questioned, manage to build and deploy complex component-based software systems all the time. How do they do it? (A similar question can be asked about reverse engineering of large, complex software systems [15].) Somehow they *must* be reasoning—most of the time correctly—about the behavior of systems using *some* informal method based on *some* informal mental models of the components' behaviors and the behaviors of their

compositions. The Invariance Theorem [14] implies that every describable object (e.g., a particular model of component behavior, a particular argument about why a program works correctly, a method for reasoning about program behavior) has an intrinsic complexity that is independent of the means of description. So, whatever models and reasoning methods allow software engineers to develop large, complex systems now, if they can be described in the usual natural language of discourse of software professionals then they can be described in the formal language of mathematics without substantial impact on the complexity of description. Perhaps the *notations* involved in the formal models and reasoning method will be initially unfamiliar to those who have not yet been educated to understand them. But our experience with CS1/CS2 students suggests that this is not an inherent impediment to making the approach practical.

Our claim for the scalability of formal mathematical modeling and formal natural reasoning, then, is that they are simply formalizations of the informal mental models and informal reasoning processes software professionals routinely use. If the resulting reasoning is unsound, then it just shouldn't be used! If it is sound, then in principle, encoding the same reasoning into formal mathematics need not add to the intrinsic complexity.

Our specific contribution in this area is that added confidence comes from formalizing the natural reasoning method and then establishing conditions under which it is sound. Recall from Section 6 that two prior attempts to do this resulted in formal systems that were unsound. Ultimately, the practical importance of having a sound formal reasoning approach that mirrors how people think depends heavily on the ability to support it with machine-processable notation such as we have introduced, and on verification and/or proof-checking tools. The latter present a longer-term challenge.

9 Conclusion

Mathematical modeling is essential for reasoning about component-based software. Without precise descriptions based on mathematical models, the benefits of component-based software development are unlikely to be fully realized because clients who use existing components will be unable to understand those components well enough to reason *soundly* about non-trivial programs that use them.

Perhaps this situation is tolerable if software components are to be used only for prototyping and non-safety-critical applications. But for "industrial strength" software systems where there can be serious consequences to software failures, the ability to reason soundly about software behavior is undeniably critical. The implications of unsound reasoning for productivity and quality—the very attributes component-based software is supposed to improve—are ominous. Fortunately, introductory CS students can learn to read and use specifications based on mathematical modeling and can appreciate the significance of appropriate modeling in developing correct software. With open minds, a bit of continuing education, and tool support, software professionals also should be able to under-

stand and appreciate this important technique and know how to use it to reason about software system behavior.

10 Acknowledgment

We gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants DUE-9555062 and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, from the National Aeronautics and Space Administration under project NCC 2-979, from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office, and from Microsoft Research. 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, NASA, the U.S. Department of Defense, or Microsoft.

References

1. M. Aronszajn, M. Sitaraman, S. Atkinson, and G. Kulczyski, "A System for Predictable Component-Based Software Construction," in *High Integrity Software*, V. Winter and S. Bhattacharya, eds., Kluwer Academic Publishers, 2000.
2. G. Booch, *Software Components With Ada*, Benjamin/Cummings, Menlo Park, CA, 1987.
3. D. de Champeaux, D. Lea, and P. Faure, *Object-Oriented System Development*, Addison-Wesley, Reading, MA, 1993.
4. D. Fleming, *Foundations of Object-Based Specification Design.* Ph.D. diss., West Virginia University, Dept. Comp. Sci. and Elec. Eng., 1997.
5. D.P. Freedman and G.M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products, 3rd ed.,* Dorset House, New York, 1990.
6. D.E. Harms and B.W. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Trans. on Soft. Eng.*, Vol. 17, No. 5, May 1991, pp. 424-435.
7. W.D. Heym, *Computer Program Verification: Improvements for Human Reasoning.* Ph.D. diss., The Ohio State Univ., Dept. of Comp. and Inf. Sci., 1995.
8. D. Knuth, "Knuth Comments on Code: An Interview with D. Andrews," Byte, Sep. 1996, http://www.byte.com/art/9609/sec3/art19.htm (current Oct. 11, 1999).
9. G.T. Leavens and Y. Cheon, "Extending CORBA IDL to specify behavior with Larch," *OOPSLA '93 Workshop Proc.: Specification of Behavioral Semantics in OO Info. Modeling,* ACM, New York, 1993, pp. 77-80; also TR #93-20, Dept. of Comp. Sci., Iowa State Univ., 1993.
10. R. Orfali, D. Harkey, and J. Edwards, *The Essential Distributed Objects Survival Guide,* J. Wiley, New York, 1996.
11. S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification of Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. on Soft. Eng.,* Vol. 21, No. 2, Feb. 1995, pp. 107-125.

12. M. Sitaraman, *An Introduction to Software Engineering Using Properly Conceptualized Objects.* WVU Publications, Morgantown, WV, 1997.
13. M. Sitaraman and B.W. Weide, eds., "Component-Based Software Using RESOLVE," *ACM Software Eng. Notes*, Vol. 19, No. 4, 1994, pp. 21-67.
14. J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, Elsevier Science Publishers, Amsterdam, 1990.
15. B.W. Weide, J.E. Hollingsworth, and W.D. Heym, "Reverse Engineering of Legacy Code Exposed," *Proc. 17th Intl. Conf. on Software Eng.*, ACM, Apr. 1995, pp. 327-331.
16. B.W. Weide, S.H. Edwards, W.D. Heym, T.J. Long, and W.F. Ogden, "Characterizing Observability and Controllability of Software Components," *Proc. 4th Intl. Conf. on Software Reuse,* IEEE CS Press, Los Alamitos, CA, 1996, pp. 62-71.
17. B.W. Weide, *Software Component Engineering,* OSU Reprographics, Columbus, OH, 1999.