Formal Methods Application: An Empirical Tale of Software Development

Ann E. Kelley Sobel, *Member*, *IEEE Computer Society*, and Michael R. Clarkson

Abstract—The development of an elevator scheduling system by undergraduate students is presented. The development was performed by 20 teams of undergraduate students, divided into two groups. One group produced specifications by employing a formal method that involves only first-order logic. The other group used no formal analysis. The solutions of the groups are compared using the metrics of code correctness, conciseness, and complexity. Particular attention is paid to a subset of the formal methods group which provided a full verification of their implementation. Their results are compared to other published formal solutions. The formal methods group's solutions are found to be far more correct than the nonformal solutions.

Index Terms—Formal methods, software specifications, software engineering curriculum.

1 Introduction

As part of a grant to study the integration of formal methods into a undergraduate software engineering curriculum,[1] a group of students participated in a sequence of courses that taught and used formal analysis. The goals of the grant included demonstrating the potential of undergraduate students for learning formal analysis techniques, establishing the feasibility of teaching formal analysis, and increasing the complex problem solving skills of the students. In order to establish the latter, several measures of the formal methods students' abilities were taken over the three year study for comparison to other students in the major. One of these measures, the ability to develop software, provides evidence that the formal methods students had increased complex problem solving skills and also supports the common belief of formal analysis advocates that the use of formal analysis during software development produces "better" programs [1], [8], [12]. This work provides, for the first time, concrete empirical evidence that this common belief is indeed true, using the metrics of code correctness, conciseness, and complexity to characterize a "better" program.

This paper is based on two classes of students at Miami University of Ohio that studied object-oriented design (OOD) in a one-semester course. One of the classes studied the material as it is typically taught at the university and, henceforth, will be referred to as the control group. The other class, in conjunction with the OOD course material, also studied the use and application of formal methods during software design. This class will be referred to as the formal methods group. These students had already received two semesters of instruction in formal methods: one semester on formal program specification and derivation, and one semester on the axiomatic semantics of abstract data types. The formal method that was employed uses first-order logic as a specification language and is based on that presented by Cohen [4].

As a project in the OOD course, both classes were assigned the development of an elevator system. Each of the groups was divided into teams of students, with an average of two students on each team. There were six teams in the formal methods group and 13 teams in the control group. The inspiration for the elevator problem came from a 1987 IEEE Workshop on Software Specification and Design [9]. The elevator system was to simulate the operation of an elevator in carrying passengers, and graphically display the state of the system. In addition to the functioning executable and full source code, all student teams were encouraged to submit a Unified Modeling Language (UML) [10] diagram showing their design of the system. The formal methods group was also asked to submit a formal specification, written using first-order logic, of the elevator system.

The empirical data presented in this work clearly demonstrates the benefits of formal analysis. Most remarkably, the percentage of implementations that passed a standard set of test cases was 45.5 percent of control teams versus 100 percent of formal methods teams. The formal methods group produced better designs and implementations than the control group. Further analysis of the results of this project revealed interesting details about the structure of the formal methods group's solutions.

The following sections present a comparison of the development efforts of these two groups. Section 2 provides background on the experimental curriculum. The experimental design for the elevator project is described in Section 3. The comparison begins in Section 4 with a description of the exact problem statement given to each group. It continues in Section 5 with a presentation of the various designs that the student teams produced. In

1. CDA-9522257 from the US National Science Foundation's CISE Educational Innovation program.

- *A.E.K. Sobel is with the Computer Science and Systems Analysis Department, Miami University, 230 J Kreger Hall, Oxford, OH 45056. E-mail: sobelae@muohio.edu.*
- *M.R. Clarkson is with the Computer Science Department, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501. E-mail: clarkson@cs.cornell.edu.*

Manuscript received 19 July 1999; revised 21 Sept. 2000; accepted 9 Jan. 2001. Recommended for acceptance by D. Rosenblum.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 110273.

TABLE 1
Course Sequence and Status

Course Title	New	Existing
Introduction to Program Derivation	•	
Semantics of Data Structures		•
Object-Oriented Design		•
Formal Analysis of Concurrent Programs	•	
Software Engineering		•
Practicum in Software Development		•

Section 6, the implementations of the systems are compared statistically based on metrics of correctness, conciseness, and complexity. An account of one team's fully formal solution to the problem is presented in Section 7, and compared to several published solutions from the IEEE workshop. Section 8 concludes the paper.

2 Background

The elevator project described in this paper was part of an experimental curriculum that had the goal of introducing the study of formal methods early in an undergraduate software engineering curriculum. The established curriculum for the department in which the experimental curriculum was taught is a blend of computer science, software engineering, and operations research courses. The injection of formal analysis into the curriculum occurred through two new courses and modification of four existing core courses. The two new courses dealt solely with formal methods. The sequence of courses and their status is shown in Table 1.

The elevator project occurred in the course on OOD. Thus, the students in the formal methods group had taken two previous courses involving formal methods—the program derivation and the modified data structures courses. The control group had not taken the program derivation course and their data structures course did not teach any formal semantics. Students from both groups had also taken other classes within the department since this project occurred in the first semester of most students' third (junior) year of undergraduate education.

The program derivation course was the first course offered in the experimental curriculum. Students were taught to specify programs using first-order logic and Cohen's specification notation [4] and to represent programs with Dijkstra's guarded command code [6]. Proofs were created manually and made use of Hoare triples and the wp predicate transformer [7]. The programs examined in this class were typically quite small, usually under 15 lines of code.

The data structures course taught typical abstract data types (ADTs), including lists, stacks, queues, and trees. The formal analysis component that was added to this course for the formal methods group was formal specification of the semantics of ADT operations. Students defined these operations using an algebraic notation and proved properties of programs that used these operations.

For more detailed information about the experimental curriculum and its results, see [15], [16], [19].

3 Experimental Approach

Given the goal of establishing an increase in the complex problem solving skills of students who use formal analysis, this particular experiment tests the hypothesis that the formal methods group solutions were better than the control group solutions (using the criteria of code correctness, conciseness, and complexity) due to their use of formal analysis.

The student population of the control group consisted of a random sample of Systems Analysis majors[2] at Miami University of Ohio. The students in the formal methods group were self-selected in that they volunteered to take the formal methods curriculum. Their reasons for this choice included potential interest in the topic and a desire to try something different. The experimental students took a learning style survey which categorized them as collaborative and competitive.

Despite the fact that the experimental group was self-selected, standardized tests revealed no statistical difference between the abilities of the two classes of students as of the first common course of the experimental and standard curriculum [15]. For this particular experiment, the self-selection of the formal methods group was not weakness, but, in fact, a necessity since the use of random selection to form a small group generally does not produce equivalence with a randomly selected large group [5].

By starting the experiment with two equivalent groups, it was crucial to maintain their equivalence with the exception of the continued exposure of formal analysis to the formal methods group. Every attempt was made to teach the two groups the same standard material. In courses in the sequence, both groups used the same texts and were taught by the same instructor. Both groups were given the same programming assignments and exams. Thus, prior to the elevator project described in this paper, both groups had taken the same data structures course[3] and had been taught the same material on OOD.

Since both populations were alike in all aspects except for the use of a formal method, the educational experiment

2. The major has since changed to Computer Science and Systems Analysis.
3. There may have been some small number of students in the control group that had taken a different data structures course since some students do not follow the required course sequence.

method of difference was used to establish the cause for the different outcomes of the programs generated by the two populations [3]. The conclusion can be made that it was the use of formal analysis employed by the students during the development of their software solution which caused the increase in the correctness of their solutions.

4 Requirements

Below is an excerpt from the program requirements given to both groups of students:

> You are to create a... program that allows a user to issue a set of elevator requests, floor and direction. These are entered through menus and dialogs and are displayed in the "request" view. A request contains the floor at which the request is made and also the floor to which the user wants to go. In another view, the elevator and floors of a building are graphically displayed. When the user presses the "GO" button in this graphic view, the elevator proceeds to process the requests that have been entered via the other view. The application shows the elevator going from one floor to another processing these requests. The current request, current floor, and status of elevator (stopped, moving up, moving down) should be displayed in text at the bottom of this graphic display. While the elevator is processing these requests, the user may enter other requests in the other view... [The elevator scheduling] algorithm should examine all current requests to determine the next floor and direction. When a new request is added, the algorithm should recalculate the next floor and direction.

In addition to this requirement, there was an implicit requirement of using object-oriented design principles [2], C++ [17] as the implementation language, and the Microsoft Foundation Classes (MFC) [11] for the graphical interface.

5 Design

5.1 Designs: Control Group

Since students in the control teams were not required to submit a design for their elevator systems, almost no record of the design (if any) these teams used is available. None of these teams elected to submit a UML diagram, as they were invited to do. Thirteen control teams submitted an executable with nine of those teams submitting the corresponding source code.

The source code from the control teams does reveal some insight into their design: Four of the nine teams that submitted source code used a design that was tightly coupled. That is, they mixed functionality from the graphical display and the elevator system into the same modules.

Four of the 13 teams that submitted executables also submitted pseudocode for the algorithm they used to control scheduling of the elevators. Each of the four designs revealed by the pseudocode is different—no common design appeared.

The first of these four algorithms guarantees processing of requests, but at the expense of efficiency. The elevators are scheduled to travel in a loop from the first floor to the top floor and back down again, *ad infinitum*. The second algorithm is based on the principle of least work. The elevators only change direction when it becomes fruitless to continue in the current direction. The algorithm is very complex and heavily based on case analysis of the state of the system. The third algorithm has each elevator always servicing the nearest request and that request only. The final algorithm is described incompletely. It fulfills all requests for stops originating from passengers inside the elevator before admitting new passengers. It, however, does not describe the method for deciding which passengers to admit.

5.2 Designs: Formal Methods Group

Two types of design artifacts exist for the formal methods teams. The first is voluntary submission of UML diagrams. The second is specifications in the form of preconditions, postconditions, and invariants written for particular functions in the teams' designs.

Out of the six teams, three submitted UML diagrams. In their UML diagrams, two of the teams indicated a logical abstraction between the elevator system and the GUI library in which it was being implemented. That is, the elevator classes were designed independently of the graphical interface. This is an arguably good design decision in that it reduces coupling between the elevator backend and the graphical frontend. The third team, however, produced a tightly coupled design. The modules that control the state of the elevator are mixed with those controlling the display of the elevator.

The completeness of specifications varied between the four of the six teams that submitted a specification. Three of the teams specified functions for deciding when and where to move the elevator (thus, comprising the scheduling functionality), while one team specified only the maintenance of the request lists. As an aggregate, the teams specified the following types of functions:

- Updating of the request lists
- Checking the current request in the system
- Movement of the elevator
- Updating the people and requests inside the elevator
- Changing of the elevator's direction

For example, one team specified the loading, unloading, and moving of the elevators. Their postcondition for the function that performed this operation was:

$$\begin{aligned}
&(\exists p : Person \mid OnFloor(e,p)\\
&\quad \wedge e.direction == p.direction : AddPerson(p))\\
&\quad \wedge (GoingDown(e) \Rightarrow e.current\_floor := e.current\_floor - 1\\
&\quad \vee GoingUp(e) \Rightarrow e.current\_floor := e.current\_floor + 1)\\
&\wedge (\exists p : Person \mid p.ending\_floor ==\\
&\qquad e.current\_floor : RemovePerson(p)).
\end{aligned}$$

This specification demonstrates that the team has clearly defined exactly what operations (removal, addition, changes in the current floor) should occur and in what states these should occur, but that their familiarity with the specification language is less than perfect. Both of the quantifiers should be $\forall$ rather than $\exists$. Also, the assignment and equality operators, borrowed from guarded command code and C++, respectively, are inappropriate for use in first-order logic. Only the Boolean equality operator $(=)$ should be used. Finally, their specification leaves a part of the state space unexamined. If the elevator is neither going up nor down, but is instead halted, the elevator will never move again, according to this specification. Other than this last error, the specification captures the elevator movement quite well.

Another team specified the movement of the elevator in terms of a target destination floor:

$$\begin{aligned}&CurrDest = CurrFloor \Rightarrow\\&\quad(\neg IsEmpty(e) \wedge CurrDest = FarthestAwayRequest(e))\\&\quad\vee (IsEmpty(e) \wedge (\neg SysQueueIsEmpty \wedge CurrDest =\\&\qquad NearestSysRequest(e)\\&\qquad\quad \vee SysQueueIsEmpty \wedge CurrDest = 1)).\end{aligned}$$

This team used the specification language correctly and was able to clearly and concisely specify the changing of the current destination of the elevator. Whether this particular algorithm is a good scheduler is not immediately apparent, but the effects of the function have been captured well.

Three of the formal methods teams also provided pseudocode descriptions of their scheduling algorithm, similar to the control teams. The first is a case analysis of what direction to move, but without any knowledge of a global list of requests in the system. The elevator must move to a floor to discover if any requests exist on it. The second is a FIFO queue of requests, with the modification that any requests that could be fulfilled enroute to the current request being serviced are also processed. The final algorithm is based on maintaining, instead of a current direction for the elevator (as did all other algorithms), a target destination floor. The destination is recalculated based on the passengers inside the elevator and the outside requests.

5.3 Comparisons

The OOD's used by the control and formal methods groups were not significantly different. The control group did produce a greater percentage of coupled designs, but it is unclear whether this is a result of the formal methods group's training in formal analysis. The algorithms for scheduling that the groups used also seem to vary widely. Only one of the four nonformal algorithms is reasonable in that it is at least mindful of efficiency and service of all requests. Two out of the three formal algorithms do in fact consider efficiency and liveness (service of all requests). This would suggest that formal analysis training increased the students' ability to design algorithms.

6 Implementation

Several metrics were used in evaluating the implementations submitted by the teams. The most important consideration was the functional correctness of the program. In order to measure this, a set of test cases was developed that addressed six different scenarios in the elevator system. An executable had to pass all six cases in order to be considered correct. Conciseness of the code was measured by lines of code, broken into two categories (request maintenance and elevator scheduling). Complexity of the code was measured by the number of loops, selection statements, and maximal nesting depth. Finally, style was observed qualitatively.

TABLE 2
Statistics from the Control Teams' Implementations

Statistics from the control teams' implementations			
	Avg.	Std. dev.	Conf. int. ($\alpha = .05$)
Total lines	136	88	(82, 191)
Request	67	83	(15, 119)
Scheduling	69	39	(45, 93)
Loops	7	6	(3, 11)
ifs	16	10	(10, 22)
Cases	30	20	(17, 42)
Deepest nesting	4	1	(4, 5)

6.1 Implementations: Control Group

The majority of the control teams developed implementations that were functionally incorrect. Three implementations were completely nonfunctional and failed all the test cases—their elevators would not move, or would stop at floors randomly, without any of the conventional functionality of an elevator. Three more implementations failed at least one of the test cases. Two of the implementations were not available for testing in that no executable existed and the executable could not be built using locally available libraries. This left only five out of 11 implementations, or 45.5 percent, as correct.

Conciseness and complexity varied greatly among the control implementations. Nine sets of source code from the 13 teams were available for analysis. The summary statistics are shown in Table 2.

The style of the code produced by the control teams was almost uniformly poor. In only two cases was the code for elevator scheduling and request maintenance encapsulated outside of the interface classes. Four of the nine implementations mixed their code for maintenance and scheduling, in one case into a single, very large function. Several examples were found of heavy case analysis. Documentation was nearly nonexistent. Furthermore, several teams duplicated code instead of creating functions, or using arrays and loops. For example, one team hard-coded each of the elevators in their system as a separate variable, and duplicated code for the scheduling, changing only the name of the variable each time.

An example of the typical, and especially the poorer, code written by the control teams would be prohibitively long to reproduce here. However, an example of the better code written by one team gives an idea of the nature of the code written by the rest of the teams. One of the better elevator scheduling algorithms that was submitted by the control teams is shown in Figs. 1a and 1b with the comments stripped to conserve space.

The fact that this is relatively good code for the control teams raises some alarm. It is a large function and very similar code is repeated three times within it. It is nested rather deeply. Even with the 16 lines of comments that were stripped from it, it is still not easy to read or to understand.

```
void CElevatorDoc::ProcessFloorStop()
{
   char TempState;
   bool found = false;
   bool found1 = false;
   int NextDestination;

   TempState = 'S';

   Destinations[CurrentFloor].RemoveAll();

   if (State == 'U')
   {
      for (int count = CurrentFloor; count<=4; count++)
      {
      if (!Destinations[count.IsEmpty())
         TempState = 'U';
      }

      for (count = CurrentFloor; count<=4; count++)
      {
         POSITION pos;
         pos = Requests[count].GetHeadPosition();

         while ((pos != NULL) && (found == false))
         {
            if ((Direction(count, Requests[count].GetNext(pos)))
                == 'U')
            {
                found = true;
                TempState = 'U';
            }
         }
      }
   }
   else if (State == 'D')
   {
      for (int count = CurrentFloor; count >= 0; count--)
      {
         if (!Destinations[count].IsEmpty())
            TempState = 'D';
      }

      for (count = CurrentFloor; count >= 0; count--)
      {
         POSITION pos;
         pos = Requests[count].GetHeadPosition();

         while ((pos != NULL) && (found == false))
         {
            if ((Direction(count, Requests[count].GetNext(pos)))
                == 'D')
            {
                found = true;
                TempState = 'D';
            }
         }
      }
   }
```

Fig. 1a. Code sample from one of the control group teams.

It can be concluded from the above that the control teams performed badly in this programming exercise. Their code was extremely incorrect and made complex by the use of case analysis. Their coding style was poor. Overall, the control implementations showed that the students lacked the ability to program well.

6.2 Implementations: Formal Methods Group

The formal methods teams' implementations were all functionally correct—all six of the implementations passed all six test cases. As with the control teams, however, conciseness and complexity varied widely. All six sets of source code were available for analysis, producing the

```
    State = TempState;

    if (State == 'S')
    {
       for (int count = 0; count<=4; count++)
       {
          if ((!Requests[count].IsEmpty()) && (found1 == false))
          {
             if (CurrentFloor == count)
             {
                 POSITION pos;
                 pos = Requests[count].GetHeadPosition();
                 while ((pos != NULL) && (found == false))
                 {
                    NextDestination = Requests[count].GetNext(pos);
                    TempState = Direction(count, NextDestination);
                    found1 = true;
                 }
             }
             else
             {
                 TempState = Direction(CurrentFloor, count);

                 found1 = true;
             }
          }
       }
    }

    State = TempState;
    ExchangePeople(CurrentFloor, State);
    UpdateAllViews(NULL);
}
```

Fig. 1b. Code sample from one of the control group teams (cont.).

statistics shown in Table 3. The code that the formal methods teams produced was in some ways similar to that of the control group. Two of the teams failed to encapsulate their request maintenance and elevator scheduling code from the interface code. Of the remaining four, two mixed most of these two types of code into one large function. The final two teams produced an implementation that was both encapsulated and modular. Two teams made use of heavy case analysis in their scheduling algorithms. Documentation was at least present in most of the teams' code, but was not particularly verbose.

One team produced an elevator scheduling algorithm that was atypically poor compared to the rest of the teams. Their algorithm consisted of one large, 128-line function that contained hard-coded, though identical, algorithms for the separate cases of one, two, and three active elevators. Again, this was an atypical case and an outlier that greatly influenced the above statistics.

As an example of one of the better implementations from the formal methods teams, consider the code presented in Figs. 2a and 2b. It is based on the idea of servicing requests as a FIFO queue, but also will service any compatible requests along with its primary service goal from the head of the queue. Once again, comments have been stripped.

This sample of code illustrates several of the points made above. It is not necessarily more concise than that of the example from the control group (Section 6.1), but it is less complex. In particular, it uses less nesting and loops. It exhibits more encapsulation in the member functions that are called on request and elevator objects. It does repeat a certain amount of code, thus it could be made more modular.

In summary, the formal methods teams all produced an implementation that was correct. However, their code was not particularly concise, nor did it avoid high levels of complexity in some cases. The style of their code was mixed—some teams exhibited good style, while others did not. The formal methods teams performed well, overall, although the actual code they wrote could be improved.

6.3 Comparisons

The most important comparison that can be made between the groups is in the category of correctness. Functional correctness, as defined by the six standard test cases to which all the groups' programs were subjected, was achieved by only 45.5 percent of the control groups and 100 percent of the formal methods groups. This is a phenomenal example of how training in formal analysis can benefit programmers. It could be claimed that another factor besides this training affected the results, such as a student's intelligence or programming experience. However, empirical data in [15] suggests that no significant difference in intelligence, as measured on standardized tests, existed in the student population. In addition, the

TABLE 3
Statistics from the Formal Methods Teams' Implementations

Statistics from the formal methods teams' implementations			
	Avg.	Std. dev.	Conf. int. ($\alpha = .05$)
Total lines	117	44	(81, 154)
Request	45	38	(14, 76)
Scheduling	72	38	(41, 104)
Loops	4	5	(0, 9)
ifs	16	10	(7, 24)
Cases	24	11	(14, 33)
Deepest nesting	5	3	(2, 7)

students had all taken the same sequence of programming classes—the only meaningful difference being the incorporation of formal methods into one track. The conclusion left, then, is that training in formal analysis gave students an increased ability to create functionally correct programs.

The categories of conciseness and complexity show much less differentiation between the formal methods and control groups. In each of the seven metrics reported above, there exists no statistically significant difference ($\alpha = .05$) between the two groups. Remarkably, by increasing to $\alpha = .2$ and removing the outlier from the formal methods group discussed above (Section 6.2), nearly all complexity metrics become significant, as illustrated in Table 4.

These results are puzzling. It has been suggested that the use of formal analysis leads to increased conciseness and decreased complexity [1], [8], but this was not observed at any high level of confidence. The only positive result is that decreased complexity was observed in two-thirds of the formal methods teams, albeit at a low level of confidence. But, at this same confidence level, the lines of code devoted to the scheduling algorithm became statistically identical, leading to the conclusion that there was no increase in conciseness.

In answer to this puzzle, we suggest that the full benefits of formal analysis (including at least correctness, conciseness, and simplicity) can be fully realized only when the entire formal method is applied. The formal methods teams

```
void CFloorsView::ProcessRequests(CDC* pDC)
   CFloorsDoc* pDoc = GetDocument();
   CRequest request;
   if(!pDoc->m_requests.IsEmpty())
   {
      request = pDoc->m_requests.GetHead();
      pDoc->m_elevators[0].SetDestFloor(request.GetCurFloor());

      if(pDoc->m_elevators[0].GetCurFloor()
         < pDoc->m_elevators[0].GetDestFloor())
         pDoc->m_elevators[0].SetMotion("Moving Up...");
      else if (pDoc->m_elevators[0].GetCurFloor()
         > pDoc->m_elevators[0].GetDestFloor())
         pDoc->m_elevators[0].SetMotion("Moving Down...");

      MoveElevator(pDoc->m_elevators[0].GetCurFloor(),
         pDoc->m_elevators[0].GetDestFloor());
      pDoc->m_elevators[0].SetMotion("Elevator stopped.");
      process = 0;
   }
}

void CFloorsView::WatchRequests(int curMoveFloor,
   int destMoveFloor)
{
   CFloorsDoc* pDoc = GetDocument();

   POSITION pos1;
   CRequest request;

   request = pDoc->m_requests.GetTail();
   pos1 = pDoc->m_requests.GetTailPosition();

   int sizelist = pDoc->m_requests.GetCount();
```

Fig. 2a. Code sample from one of the formal methods teams.

```
  for(int i = 1; i < sizelist; i++)
  {
    if(pDoc->m_elevators[0].GetMotion() == "Moving Up...")
      if (curMoveFloor == request.GetCurFloor() &&
        destMoveFloor >= request.GetDestFloor() &&
        request.GetDirection() == "Moving Up...")
      {
        pDoc->m_elevators[0].SetMotion("Elevator stopped.");
        m_stopOnFloor[request.GetDestFloor()] = 1;

        pDoc->m_requests.RemoveAt(pos1);
      }
    }

    if(pDoc->m_elevators[0].GetMotion() == "Moving Down...")
      if (curMoveFloor == request.GetCurFloor() &&
        destMoveFloor <= request.GetDestFloor() &&
        request.GetDirection() == "Moving Down...")
      {
        pDoc->m_elevators[0].SetMotion("Elevator stopped.");
        m_stopOnFloor[request.GetDestFloor()] = 1;

        pDoc->m_requests.RemoveAt(pos1);
      }
    }

    request = pDoc->m_requests.GetPrev(pos1);
  }

  if(m_stopOnFloor[curMoveFloor])
  {
    pDoc->m_elevators[0].SetMotion("Elevator stopped.");
    Invalidate();
    m_stopOnFloor[curMoveFloor]=0;
  }
}
```

Fig. 2b. Code sample from one of the formal methods teams (cont.).

stopped application of the method before deriving code. As a result, they achieved correctness (their specifications and training at least provided that), but not the other two benefits. Without formally producing code, they could only produce as good of code as others at their level of programming expertise.

The final category, style, showed another positive impact of formal methods. The style of the control teams was uniformly poor, while that of the formal methods teams varied from poor to good. This again supports training in formal analysis.

TABLE 4
Confidence Intervals on Control and Formal Methods Metrics, $\alpha = .2$

	Control	Formal Methods
Total lines	(116, 183)	(109, 134)
Request	(51, 113)	(33, 75)
Scheduling	(52, 83)	(52, 82)
Loops	(5, 10)	(1, 4)
ifs	(14, 21)	(11, 15)
Cases	(28, 42)	(21, 24)
Deepest nesting	(4, 5)	(3, 4)

7 A Fully Formal Solution

After the class assignment was finished, four students of the formal methods group undertook a more thorough formal analysis of the elevator system. This team took the specification of the most important part of the system (the elevator scheduling algorithm), implemented it using guarded command code (GCC), verified that the code satisfied the specification, and, finally, translated the GCC into C++.[4] This team will be referred to as the verification team throughout this section.

7.1 Requirements

In addition to the requirements in Section 4, the verification team chose to gather further requirements for the scheduling algorithm. Their informal definition of the elevator scheduling problem was taken from the call for papers of

4. This work was presented in a poster session at the 1998 SIGCSE conference [14]. Special thanks to Toni Lehmkuhl, Stephanie Taylor, and Bryce Williams for allowing part of that work to be reproduced here.

the 1987 IEEE International Workshop on Software Specification and Design, where it was explicitly proposed as a "challenging exercise" [9]. The informal requirements were adapted as follows:

1. Maintain a set of buttons that have been pushed inside the elevator (requested floor number).
2. Maintain a set of buttons that have been pushed outside the elevator (floor number on which a passenger is waiting and the requested direction).
3. Halt when the elevator has no more requests to service.
4. Service all outside requests eventually with all floors given equal priority.
5. Service all inside requests eventually with all floors being serviced sequentially in the direction of travel.
6. Ensure that requests are processed in a timely manner and that the direction of elevator movement occurs in a logical manner that maximizes the use of the elevator.

7.2 Design

The verification team produced both a UML diagram and formal specifications.

The primary classes indicated in their UML diagram were *Elevator* and *ElevatorSystem*. The team concluded that the majority of the methods in these classes needed little in the way of algorithmic design, for example, list operations. The scheduling algorithm itself was encapsulated in the *UpdateElevator* method of class *Elevator*. This method was the focus of their verification.

The guiding principle behind their *UpdateElevator* method is that of *least work*; in particular, the elevator does not change direction until there is no reason to continue in the current direction. By continually traveling in one direction for a maximal distance and only taking on passengers traveling in that direction, inside requests can be serviced in a timely manner.

After the members of the team produced a first-order specification of the method, they wrote guarded command code based on that specification. They then produced proofs that the guarded command met its specification. For examples of the specification, guarded command code, and proof, see the appendix.

The original specification was revised as errors were discovered through the verification process. The errors were generally due to the specification being incomplete. For example, one error resulted from not specifying how the elevator should resume motion after halting (as did another team in an above example). Another important result of the verification was the discovery of an error that led to the generation of an abort statement.

The original draft of the specification required approximately five hours for the team to write. Revising the specification to account for errors required approximately ten additional hours, some of which were used for generating portions of the verification. The verification itself required another ten hours to complete.

A key result of the formal method used by the verification team is that all errors in the specification were discovered before implementing the code—that is, the specification never required revision due to errors discovered during actual code testing. The verification process produced a sound design that did not need changes based on discoveries from implementation and testing.

TABLE 5
Values from the Verification Team's Implementation

Values from the verification team's implementation	
Total lines	105
Request	11
Scheduling	94
Loops	7
ifs	15
Cases	24
Deepest nesting	4

7.3 Implementation

In order to implement the functions which were formally derived, the verification team translated their guarded command code into C++ in a very direct manner. Predicates were translated as function calls, multiple assignments were changed into sequences of single assignments, and nondeterminism in if statements was resolved in the order in which the guards appeared. The resulting code is given in Appendix A.4.

The formal methodology applied by the verification team proved beneficial in implementing the code for this algorithm. The first run of the program was not error-free, but the errors encountered were related exclusively to the code not derived from a specification. *ReasonToGo* was the only procedure involved in the scheduling algorithm itself that contained an error in logic, and it was, not coincidentally, the only procedure in the algorithm that was not formally derived. Further testing of the code showed the sections generated by the application of formal methods to be correct, not just in theory, but in practice.

The verification team's implementation passed all six test cases that were used to evaluate the control and formal methods teams' implementations, thus making it functionally correct. Its conciseness and complexity are characterized by the values in Table 5. Note that the values in Table 5 include more code than is presented in the appendix, thus increasing the length and complexity of the code.

The style of the verification team's code is exceptionally good. It is both encapsulated and modular. The use of case analysis is avoided. Documentation was present both in the comments in the source code and in the form of a complete specification and guarded command code.

To summarize, the verification team's implementation was functionally correct, just as the rest of the formal group's implementations. Even more importantly, it was virtually error-free at the first run of the program. Comparisons based on conciseness and complexity are difficult to make because of the small sample size (only one). Finally, the style was better than all the other teams. The verification team performed very well in this programming exercise.

7.4 Comparisons to Published Formal Solutions

Many solutions were offered to the elevator scheduling problem as presented at the IEEE Workshop on Specification and Design. Other solutions to the problem dealt primarily with the physical requirements of the elevator. One concentrated on elevator lights (on/off), doors (open/closed), and buttons (press/release) [18, p. 266]. Another included the operation of the emergency button in the design [13, p. 23]. All of these aspects are important in the physical operation of an elevator, but ignore the essence of designing an algorithm to control the movement of the elevator itself. Furthermore, these solutions did not include in their specifications the maintenance of request sets for inside and outside the elevator. Tracking such information is crucial in designing an elevator that will minimize waiting time for passengers.

One published solution to this problem [13, p. 26] gives priority to the passengers in the elevator and those waiting on floors that the elevator passes by, not allowing the elevator to check for requests on other floors. If no one in the elevator ever requests a floor where people are waiting and the elevator never passes that floor, then those people will never be picked up. This ignores a serious part of the functionality of an elevator by not guaranteeing that all requests are eventually serviced.

The solution to the elevator scheduling problem developed by the verification team is more logical in nature, focusing on the actual algorithm that determines the direction the elevator should travel dependent upon the requests of people waiting for and riding inside the elevator. This quintessential part of the problem was captured mathematically in a manner that permits proof of the correctness of the code derived from it. Furthermore, the solution is much more complete and detailed than others presented at the IEEE Workshop.

8 Conclusions

The development of an elevator scheduling system by three groups of teams of students, with each group utilizing formal analysis to a different extent, revealed interesting details about the benefits of formal analysis.

For a requirements definition, the control and formal methods groups were content with the somewhat vague definition given to them. Only the verification team, which utilized the formal method to its full extent, sought a much more detailed requirements definition.

In the design phase of the project, the control teams did not produce any artifacts that can be analyzed, but their source code shows a lack of characteristics that would result from good design. In particular, their code exhibited a high degree of coupling with the functionality of the interface code. The designs of the formal methods group were at least documented in diagrams and first-order specifications, although some of these were incomplete. Their designs showed less coupling and their specifications of elevator scheduling and maintenance request functions adequately capture the effects of these functions, even though their understanding of the specification language showed some deficiencies. The verification team's design was very complete, including a class diagram of the entire system, as well as a full specification of the elevator scheduling algorithm.

The implementations produced by the three groups demonstrated that the application of formal analysis provided great benefits. The most important result was the percentage of implementations that passed a standard set of test cases: 45.5 percent of control teams versus 100 percent of formal teams. Beyond correctness, there appears to be a certain advantage in decreased complexity of code for the formal teams, but not at a high level of confidence. Conciseness did not seem to be affected by the use of formal specifications. This fact may be explained by the formal methods group only using formal analysis to write specifications and not actually deriving their code. The verification team did indeed achieve significant levels of conciseness; however, their one data point is not sufficient for a comparison of conciseness or complexity. Finally, a positive correlation appeared to exist between the increased use of formal methods and better coding style.

These results support the hypothesis that the use of formal analysis led to better solutions by the formal methods group. The enhanced ability to develop software is a demonstration that the formal methods group had increased complex problem solving skills. This validates the common belief of formal analysis advocates that the use of formal analysis during software development produces "better" programs. It is our hope that these results contribute to the inclusion of formal analysis in software engineering curricula at other universities.

Appendix A
Formal Verification

A.1 Specification

The parameters required by $UpdateElevator$ are:

- $CurFloor$ An integer representing the floor where the elevator is currently located
- $CurDir$ A value in the set $\{UP, DOWN, HALT\}$ that represents the direction in which the elevator is currently moving
- $InsideReqSet$ A set of integers representing the floor numbers associated with the buttons currently illuminated inside the elevator
- $OutsideReqSet$ A set of $OutsideReq$ values (pairs consisting of a floor number and direction value) that represent the set of buttons currently illuminated outside the elevator. Elements are added to this set by a process external to the elevator scheduler.

The precondition is necessary for saving the initial values of $CurDir$, $InsideReqSet$, and $OutsideReqSet$ since these values can change during the execution of $UpdateElevator$, and both the initial and final values must be referenced in the postcondition. Thus, references to CD, I, or O in the postcondition are actually references to the initial values of $CurDir$, $InsideReqSet$, and $OutsideReqSet$.

The postcondition was separated into one main portion, two predicates, and one function. These predicates, whose definitions follow, are textual substitutions for frequently repeated pieces of the postcondition, and are used to improve readability.

The only function, $Rev(Dir)$, returns the reverse of the direction Dir. For example, $Rev(UP)$ would return the direction value $DOWN$ and $Rev(HALT) = HALT$.

$ReasonToGo(Dir)$ is a predicate that is $true$ if there is a reason to go in the direction Dir. For example, $ReasonToGo(UP)$ is $true$ either if someone inside the elevator has pushed a button requesting to go to a floor higher than the floor the elevator is currently on, or if someone on a higher floor outside the elevator has pushed a button to request the elevator.

$PickupDropoff(Dir)$ is a predicate that specifies the changes that occur in the inside and outside request sets as passengers are picked up and dropped off at different floors throughout the building. When conditions permit the fulfillment of a request, this predicate specifies that the request will no longer exist in the appropriate request list (though it may be added to the other list) after execution of the algorithm. This fulfills the maintenance component of the first two English requirements.

In the main postcondition, $ReasonToGo$ is used to realize the liveness properties of the fourth and fifth requirements. As long as requests exist, this predicate will be $true$ and, thus, the elevator will not stop moving and servicing requests, although no formal proof of this has yet been devised. When $ReasonToGo$ becomes $false$ for all directions, the main postcondition chooses $HALT$ as the new elevator direction, thus fulfilling the third requirement. Other directions are chosen based upon $ReasonToGo$. The guiding principle behind the specification is that of *least work*; in particular, the elevator does not change direction until there is no reason to continue in the current direction. This is in order to meet the efficiency goals of the requirements. By continually travelling in one direction for a maximal distance and only taking on passengers travelling in that direction, inside requests can be serviced in a timely manner.

Specification of $UpdateElevator$

```
$\underline{var}\ CurFloor : int$
$;\underline{var}\ CurDir : direction\,\{CD = CurDir\}$
$;\underline{var}\ InsideReqSet : set\,of\,int\,\{I = InsideReqSet\}$
$;\underline{var}\ OutsideReqSet : set\,of\,OutsideReq\,\{O = OutsideReqSet\}$
$;CurDir, InsideReqSet, OutsideReqSet : \big(CD = HALT$
    $\wedge\big(ReasonToGo(UP)$
        $\wedge\ CurDir = UP \wedge PickupDropoff(UP)\big)$
      $\vee\big(ReasonToGo(DOWN)$
        $\wedge\ CurDir = DOWN \wedge PickupDropoff(DOWN)\big)$
      $\vee\big(\neg\big(ReasonToGo(UP) \vee ReasonToGo(DOWN)\big)$
        $\wedge\ CurDir = HALT\big)\big)$
  $\vee\big(CD \neq HALT \wedge (ReasonToGo(CD)$
        $\wedge\ CurDir = CD \wedge PickupDropoff(CD)\big)$
      $\vee\big(\neg ReasonToGo(CD) \wedge ReasonToGo(Rev(CD))$
        $\wedge\ CurDir = Rev(CD) \wedge PickupDropoff(Rev(CD))\big)$
      $\vee\big(\neg\big(ReasonToGo(CD) \vee ReasonToGo(Rev(CD))\big)$
        $\wedge\ CurDir = HALT \wedge PickupDropoff(HALT)\big)\big)$
```

where

$$Rev(Dir) \equiv \begin{cases} UP & Dir = DOWN \\ DOWN & Dir = UP \\ HALT & Dir = HALT \end{cases}$$

```
$ReasonToGo(Dir) \equiv \big(\exists i | 0 \leq i < \#I :$
        $\big(I.i > CurFloor \wedge Dir = UP\big)$
    $\vee\big(I.i < CurFloor \wedge Dir = DOWN\big)\big)\big)$
  $\vee\big(\exists o | 0 \leq o < \#O : \big(O.o.floor > CurFloor \wedge Dir = UP\big)\big)$
    $\vee\big(O.o.floor < CurFloor \wedge Dir = DOWN\big)$
    $\vee\big(O.o.floor = CurFloor \wedge Dir = O.o.Dir\big)\big)$
$PickupDropoff(Dir) \equiv \big((CurFloor, Dir) \notin O$
    $\wedge\big(C_{00} : CurFloor \notin I \vee \big(CurFloor \in I$
    $\wedge InsideReqSet = I - CurFloor\big)\big)\big)$
  $\vee\big((CurFloor, Dir) \in O \wedge \big(CurFloor \notin I$
    $\wedge InsideReqSet = I + GetDests()$
    $\wedge OutsideReqSet = O - (CurFloor, Dir)\big)$
    $\vee\big(CurFloor \in I \wedge InsideReqSet = I + GetDests()$
    $- CurFloor, OutsideReqSet = O - (CurFloor, Dir)\big)\big)\big)$
$Direction = \{UP, DOWN, HALT\}$
$OutsideReq = (floor : int,\ dir : Direction)$
```

$GetDests()$ is a call to the user interface. This function returns the set of buttons just pushed inside the elevator as passengers enter. The equivalent gathering of buttons pushed outside the elevator occurs external to this scheduling algorithm.

Specification of $ProgPD$

$\{true\}\ ProgPD(Dir)\ \{PickupDropoff(Dir)\}$

A.2 Guarded Command Code

The following guarded command code was written by the verification team based on the above specification. The code for $UpdateElevator$ and $ProgPD$, a helper function, is presented here.

Guarded Command Code for $UpdateElevator$

```
$\underline{if}\ CD = HALT\ \rightarrow$
  $\underline{if}\ ReasonToGo(UP) \rightarrow$
      $CurDir := UP; ProgPD(UP)$
    $\Box\ ReasonToGo(DOWN) \rightarrow$
      $CurDir := DOWN; ProgPD(DOWN)$
    $\Box\ \neg\big(ReasonToGo(UP) \vee ReasonToGo(DOWN)\big) \rightarrow$
      $\underline{skip}$
    $\underline{fi}$
  $\Box\ CD \neq HALT\ \rightarrow$
    $\underline{if}\ ReasonToGo(CD) \rightarrow$
      $ProgPD(CD)$
    $\Box\ \neg ReasonToGo(CD) \wedge ReasonToGo(Rev(CD)) \rightarrow$
      $CurDir := Rev(CD); ProgPD(Rev(CD))$
    $\Box\ \neg\big(ReasonToGo(CD) \vee ReasonToGo(Rev(CD))\big) \rightarrow$
      $CurDir := HALT; ProgPD(HALT)$
    $\underline{fi}$
  $\underline{fi}$
$\{R\}$
```

Guarded Command Code for $ProgPD$

```
$\underline{if}(CurFloor, Dir) \notin O \rightarrow$
    $\underline{if}\ CurFloor \notin I\ \rightarrow\ \underline{skip}$
    $\Box\ CurFloor\ \in I\ \rightarrow InsideReqSet := I - CurFloor$
```

```
        fi
    □ $(CurFloor, Dir) \in O \;\rightarrow$
        if $CurFloor \notin I \;\rightarrow$
            $InsideReqSet, OutsideReqSet := I + GetDests(),$
                $O - (CurFloor, Dir)$
        □ $CurFloor \in I \;\rightarrow$
            $InsideReqSet, OutsideReqSet := I + GetDests() -$
                $CurFloor, O - (CurFloor, Dir)$
        fi
fi
```

A.3 Proof

The verification team also produced proofs that the above guarded command met its specification. The following is an excerpt from a paper presented at [14] that shows the annotated specification, guarded command code, and proof of $ProgPD$:

Specification

$\{Q : true\}\, ProgPD(Dir)\, \{PickupDropoff(Dir)\}$

Guarded command code

```
$\{P : true\}$
if $C_0 : (CurFloor, Dir) \notin O \rightarrow$
        $T_0$ : if $C_{00} : CurFloor \notin I \;\rightarrow\; T_{00}$ : skip
        [] $C_{01} : CurFloor \in I \rightarrow T_{01} : InsideReqSet :=$
            $I - CurFloor$
        fi
    [] $C_1 : (CurFloor, Dir) \in O \;\rightarrow$
        $T_1$ : if $C_{10} : CurFloor \notin I \;\rightarrow$
            $T_{10} : InsideReqSet, OutsideReqSet :=$
              $I + GetDests(),$
                $O - (CurFloor, Dir)$
        [] $C_{11} : CurFloor \in I \;\rightarrow$
            $T_{11} : InsideReqSet, OutsideReqSet :=$
              $I + GetDests() -$
                $CurFloor, O - (CurFloor, Dir)$
        fi
fi
$\{U : PickupDropoff(Dir)\}$
```

Proof

(1) Disjunction of guards

$C_0 \vee C_1$

$\equiv<$ Substitution $>$

$(CurFloor, Dir) \notin O \vee (CurFloor, Dir) \in O$

$\equiv<$ Definition of $\notin$ $>$

$\neg((CurFloor, Dir) \in O) \vee (CurFloor, Dir) \in O$

$\equiv< \neg p \vee p >$

$true$

$C_{00} \vee C_{01} \equiv C_{10} \vee C_{11} \equiv true$ by similar logic.

(2) $\{C_{00}\}\, T_{00}\, \{U_0\}$

$wp.\underline{\text{skip}}.U_0$

$\equiv<$ Definition of $\underline{\text{skip}}$ $>$

U_0

$\equiv<$ Assume C_{00} $>$

$true$

(3) $\{C_{01}\}\, T_{01}\, \{U_0\}$

$wp.InsideReqSet := I - CurFloor.U_0$

$\equiv<$ Assume C_{01} $>$

$wp.InsideReqSet := I - CurFloor.InsideReqSet = I - CurFloor$

$\equiv<$ Definition of $:=$, $x = x$ $>$

$true$

Proofs of T_{10} and T_{11} are identical in shape, with appropriate subscripts.

(4) $\{C_0\}\, T_0\, \{U\}$

$wp.T_0.U$

$\equiv<$ Definition of $\underline{\text{if}}$ $>$

$(C_{00} \;\vee\; C_{01}) \wedge \{C_{00}\}\, T_{00}\, \{U_0\} \wedge \{C_{01}\}\, T_{01}\, \{U_0\}$

$\equiv<$ (1), (2), (3) $>$

$true$

Proof of T_1 proceeds indentically, using appropriate subscripts.

(5) $\{P\}\, ProgPD\, \{U\}$

$wp.ProgPD.U$

$\equiv<$ Definition of $\underline{\text{if}}$, Assume appropriate guards $>$

$(C_0 \;\vee\; C_1) \wedge \{C_0\}\, T_0\, \{U_0\} \wedge \{C_1\}\, T_1\, \{U_1\}$

$\equiv< (1), (4) >$

$true$

A.4 Implementation

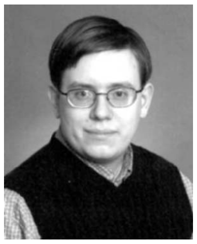
```
void CElevatorSystem::
    ProgPD(int CurFloor, int Dir)
{
  if (!InOutReqSet(CurFloor, Dir)) {
    if (InInReqSet(CurFloor)) {
      while (m_Elevator.FindStop(CurFloor)) {
        m_Elevator.RemoveStop(CurFloor);
      }
    }
  } else if (!InInReqSet(CurFloor)) {
      GetDests(CurFloor, Dir);
      RemoveRequests(CurFloor, Dir);
    } else {
      while (m_Elevator.FindStop(CurFloor)) {
        m_Elevator.RemoveStop(CurFloor);
      }
      GetDests(CurFloor, Dir);
      RemoveRequests(CurFloor, Dir);
    }
}

int CElevatorSystem::Rev(int Dir)
{
  if (Dir == UP) {
    return DOWN;
  } else if (Dir == DOWN) {
    return UP;
  } else
```

```
    return HALT;
}

int CElevatorSystem::UpdateElevator()
{
  int CD, CurFloor;

  CD = m_Elevator.GetCurrentDirection();
  CurFloor = m_Elevator.GetCurrentFloor();

  if (CD == HALT) {
    if (ReasonToGo(UP)) {
      m_Elevator.SetCurrentDirection(UP);
      ProgPD(CurFloor, UP);
    } else if (ReasonToGo(DOWN)) {
      m_Elevator.SetCurrentDirection(DOWN);
      ProgPD(CurFloor, DOWN);
    }
  } else if (ReasonToGo(CD)) {
      ProgPD(CurFloor, CD);
    } else if (ReasonToGo(Rev(CD))) {
      m_Elevator.SetCurrentDirection(Rev(CD));
      ProgPD(CurFloor, Rev(CD));
    } else {
      m_Elevator.SetCurrentDirection(HALT);
      ProgPD(CurFloor, HALT);
    }
  }
  return m_Elevator.GetCurrentDirection();
}
```

REFERENCES

[1] J.P. Bowen and M.G. Hinchey, "Ten Commandments of Formal Methods," *Computer,* vol. 28, no. 4, pp. 56–63, Apr. 1995.
[2] G. Booch, *Object-Oriented Analysis and Design with Applications,* second ed. Addison Wesley Longman, 1994.
[3] L. Christensen, *Experimental Methodology.* Allyn and Bacon, 1977.
[4] E. Cohen, *Programming in the 1990s: An Introduction to the Calculation of Programs.* Springer-Verlag, 1990.
[5] D.T. Campbell and J.C. Stanley, *Experimental and Quasi-Experimental Designs for Research.* Houghton Mifflin, 1963.
[6] E.W. Dijkstra, *A Discipline of Programming.* Prentice Hall, 1976.
[7] E.W. Dijkstra and C.S. Scholten, *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.
[8] A. Hall, "Seven Myths of Formal Methods," *IEEE Software,* pp. 11–19, Sept. 1990.
[9] IEEE, *Proc. Int'l Workshop Software Specification and Design,* 1987.
[10] G. Booch, J. Rumbaugh, and I. Jacobsen, *The Unified Modeling Language Reference Manual.* Addison Wesley, 1998.
[11] Microsoft, *Microsoft Visual C++ MFC Library Reference,* vol. 1, Microsoft Press, 1997.
[12] H. Saiedian, "An Invitation to Formal Methods," *Computer,* vol. 29, no. 4, pp. 16–30, 1996.
[13] M.D. Schwartz and N.M. Delisle, "Specifying a Lift Control System with CSP," *Proc. IEEE Workshop Software Specification and Design,* 1987.
[14] ACM, *Proc. 29th SIGCSE Technical Symp. Computer Science Education,* 1998.
[15] A.E.K. Sobel, "Empirical Results of a Software Engineering Curriculum Incorporating Formal Methods," *ACM Inroads,* vol. 32, no. 1, pp. 157–161, Mar. 2000.
[16] A.E.K. Sobel, "Emphasizing Formal Analysis in a Software Engineering Curriculum," *IEEE Trans. Education,* May 2001.
[17] B. Stroustrup, *The C++ Programming Language,* third ed. Addison Wesley, 1997.
[18] J.C.P. Woodcock, S. King, and I.H. Sorensen, "Mathematics for Specification and Design: The Problem with Lifts...," *Proc. IEEE Workshop Software Specification and Design,* 1987.
[19] http://www.eas.muohio.edu/csa/formal, 2002.

Ann E. Kelley Sobel received the PhD degree in computer science from The Ohio State University. She is an associate professor in the Computer Science and Systems Analysis Department of Miami University. Her research interests include formal methods creation and application, software engineering, and survivable systems. She is a member of the IEEE Computer Society and the ACM.



Michael R. Clarkson received the BS and BM degrees in computer science and music, respectively, from Miami University. He is currently a graduate student at Cornell University where he is working toward the PhD degree in computer science. His research interests include programming languages, formal analysis of software, and security in distributed systems.