Introducing Middle School Girls to Fault Tolerant Computing

Paolo A. G. Sivilotti and Murat Demirbas
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43221
{paolo,demirbas}@cis.ohio-state.edu

Abstract

During summer 2002, we ran a workshop module for a group of 28 eighth-grade girls. Our aim was ambitious: to introduce these students, ages 12 and 13, to computer science by focussing on the deep intellectual topic of self-stabilizing distributed algorithms and by imparting an intuitive appreciation for their use in fault tolerance. At the same time, we hoped to dispel some negative stereotypes of computer science. The module was a success according to evaluations and comments from the participants. This paper describes the sequence of exercises we developed as an elementary-level introduction to the graduate-level topics of fault tolerance and self-stabilization. We report them with the hope that others will try them in college classrooms, as we plan to do.

Categories & Subject Descriptors

K.3.2 [**Computers and Education**] : Computer and Information Science Education – *computer science education.*

General Terms

Algorithms, Design, Human Factors.

Keywords:

Distributed algorithms, Algorithm anthropomorphism, Outreach, Pedagogy.

1 Introduction

As a discipline, computer science has had limited success in attracting and retaining female students [1]. Significant projects have been launched to address this deficiency, as seen in the June 2002 issue of SIGCSE Bulletin dedicated entirely to "Women in Computer Science". Little headway, however, has been made in changing the negative stereotypes associated with the field. It is natural for recruitment efforts to target 11^{th} and 12^{th} grade students. Unfortunately, it may be too late at that point to significantly impact student perceptions of their career options. [2]

In August of 2002, the College of Engineering at The Ohio State University hosted a week-long workshop for middle school girls entering the eighth grade. The workshop, entitled "Future Engineers' Summer Camp", attracted participants primarily from the local area. The aim was to introduce the girls to a variety of science and engineering subdisciplines.

Since the participants were young, many societal prejudices about the nature of various science and engineering disciplines were less entrenched. On the other hand, working with students so young presented a challenge: Activities had to be accessible, but also relevant and honest with respect to the intellectual nature of the discipline.

As our contribution to this week-long workshop, we developed a 2.5 hour module to introduce computer science. A classic approach for introducing CS concepts is to use a simple programming language such as LOGO [4]. Students are quickly exposed to basic concepts in imperative programming, flow-of-control, testing, and debugging. Unfortunately, students must also come to grips with syntax requirements which can mask the fundamental issues.

Our approach was different. We aimed to introduce these students to CS concepts near the research frontier. In particular, we presented a distributed self-stabilizing algorithm and illustrated its use in fault tolerant computing. Thus, students glimpsed the intellectual profile of issues being addressed by researchers in computer science.

The module does not presuppose any background in computer science. (In fact, only two of the participants had any prior programming experience.) It was presented in three parts. The first part illustrated the basics of programming and software engineering through programming a small robot. The second part built on the first, but introduced the notion of parallel programming through participant animation of several algorithms. Finally, the last part extended the notion of parallel programs to motivate two algorithms for fault tolerance. By the end of the module, the girls had acted out an implementation of Dijkstra's self-stabilizing token ring algorithm [3].

The structure of this paper reflects the structure of the module. For each of the three parts, the presentation and the participant activity are described. We focus on the last part, the fault tolerance activity. The paper concludes with assessments from the participants and our own observations.

*S*IGCSE'03, February 19-23, 2003, Reno, Nevada, USA.

2 Preliminaries: Software Engineering

The nature of a program as a sequence of instructions is fundamental to the presentation of many algorithms, including self-stabilizing ones. Given that no CS background is assumed, programs must be introduced. At the same time, this introduction should not be obfuscated through syntax concerns of any particular implementation language.

2.1 Presentation

The elementary analogy of computers and programs to chefs and recipes was used throughout the module. The versatility and general-purpose nature of a chef was compared to that of a computer. The imperative and sequential nature of a recipe was compared to that of a program. Though simple, this analogy provided a nice unifying context through all three activities.

The chef analogy works well for discussing software engineering. Recipes, like programs, can be characterized by what ingredients they require (input) and what dish they produce (output). Basic requirements for a software engineer can be understood in terms of needed ingredients and final dish. Similarly, software design issues can be understood in terms of well-written recipes. Thus, a software engineer can be cast as a "recipe engineer".

2.2 Activity

As an exercise in writing programs, students were asked to program a robot. The robot was built with a light sensor permitting it to follow lines drawn in electrical tape on a white floor. The robot was set on a 6x6 grid, where two of the vertices were colored red (see Figure 1).

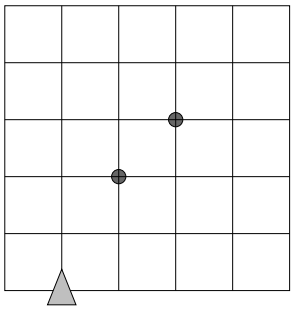

Figure 1: Grid Layout for Robot Activity

The robot had a small set of pre-programmed instructions: turn left, turn right, forward one, forward to end of grid, and take sample. The task for the students was to write a program that brought the robot from its starting position to a red vertex, had it take a sample there, then had it return to its original position. If the take sample instruction was correctly issued over a red vertex, the robot would play a little tune.

To avoid issues of syntax and compilation, each student team was given a stack of index cards. Each card had one of the robot instructions pre-printed on it. The student team wrote their program by creating a stack of cards. Due to the exceedingly simple programming language (only 5 instructions) programs were straight line (no loops) with no conditional branching. Once satisfied with their program, the students brought the stack to the teaching assistant at the computer who showed them how to use the standard graphical user interface to visually assemble blocks (corresponding to their physical cards) to form a program for the robot. The program was then downloaded to the robot and students saw the result of their efforts.

As a further challenge, students were asked to write a program that would successfully take a sample at a red vertex, regardless of where the robot was placed initially. With only a limited number of "take sample" cards (2) and no explicit conditional statements, this was quite a challenge. Nevertheless, most groups eventually discovered how to use the "go forward to end of grid" instructions to accomplish the task.

The robots were built from off-the-shelf Lego Mindstorm^{TM} components. The procedures for following grid lines and for performing each of the 5 high-level instructions are available at URL `http://www.cis.ohio-state.edu/~paolo/FESC02`. The students worked in groups of four and there were 2 separate grids and lego robots available. This activity required an hour to complete.

3 Preliminaries: Parallel Programming

The second activity introduced parallel algorithms. This activity was adapted from the description in [5], so it is only summarized briefly here.

Continuing the analogy of programs to recipes, it is easily observed that multiple chefs will result in a recipe being completed earlier. The analogy also illuminates issues of scaling and sequential bottlenecks. As an application of these ideas, three sorting algorithms were presented: bubblesort, even-odd transposition, and radix sort. The first is sequential, while the others are parallel.

Students were then arranged in groups of 10. Each group, arranged single file, represented an array of integers needing to be sorted. Each student held a number and they collectively acted out the three different algorithms. This exercise dramatically illustrates the performance gain of parallel over sequential programming.

In [5], this activity was developed for high school seniors. We found that the activity is appropriate for middle school children with very little modification. This activity was completed in 45 minutes.

4 Introducing Fault Tolerance

The third and final activity was the introduction of fault tolerance. Again, the analogy of chefs and recipes was useful for presentation. The nature of a fault, and its impact on a computation, is easily seen in the context of a chef preparing a dish. Students suggested a variety of things that could go wrong in the kitchen, and these problems were then related to hardware or software faults, both internal and environmental. While few students had written a program before this module, many had experienced, as users, the effects of faults in a computer system (*e.g.*, operating system crashes). With the cooking analogy, they saw the wide variety of possible faults and how these faults can propagate.

The parallel programming activity leads nicely into the fault tolerance discussion. Having just seen the utility of multiple "chefs in the kitchen" for performance, students quickly sug-

gested this strategy as a solution for fault tolerance. Indeed, the basic technique of modular redundancy in computer systems emerged quite naturally from the previous activity on parallel programming.

5 Fault Tolerant Token Ring

Armed with a basic appreciation for the nature of faults and possible strategies for tolerating them, the discussion turned to self-stabilization. This activity involved two algorithms for token-based mutual exclusion in a ring. The first was brittle, while the second was self-stabilizing. By animating the algorithms, students gained an appreciation for the effect of self-stabilizing distributed algorithms as well as some intuition for how these algorithms work.

5.1 Simple Token Ring

The discussion began by describing and motivating mutual exclusion. A simple strategy for coordinating a collection of processors by passing a token is easily understood by students at this level. A physical token, handed from student to student (or "chef" to "chef"), sufficed to illustrate the basic idea. With the introduction of a fault (the loss of the token), the shortcoming of this algorithm was quickly seen: Once the token is lost, it is never replaced.

5.2 Binary State Token Ring

To avoid the difficulty of losing the token, a binary state ring can be used. The single token is replaced by each processor keeping a binary variable. This variable was introduced as encoding whether or not the processor has held the token in the past.[1] If the variable for processor p has value 0, but p's left neighbor's variable has value 1, then p now has the token. Processor p can access the critical resource, and then modifies its variable to have value 1. In this way, the token proceeds around the ring.

When the token has made a full cycle around the ring, all processors will have value 1. In order for it to continue circulating, therefore, the initial processor must follow a slightly different rule than the others: If its variable has value 1, and its left neighbor *also* has value 1, then the initial processor has the token. After accessing the critical resource, The initial processor changes its value to 0. A wave of 0's is then propagated around the ring.

All processors (except the initial one) are thus following a simple rule:

> If my left neighbor's value is ever *different* from mine, I use the resource, then *copy* their value.

The initial processor has a modified rule:

> If my left neighbor's value is ever the *same* as mine, I use the resource, then makeD my value *different*.

To illustrate this algorithm, 14 volunteers were chosen. Desks were arranged in a circle, and each volunteer was given: (i) a piece of paper with a large 0 and 1, (ii) a marker to be place on either the 0 or the 1, and (iii) a one-note xylophone tone bar and mallet. It is important that the numbers (0 and 1) and the marker be large enough to be easily visible by a neighbor. It is also important that the marker be easily movable, to ensure that the token circulates quickly.

The paper and marker were used to represent state, while playing the tone bar represented using the critical resource (*i.e.*, having the token). The notes were arranged in a major scale, ascending and descending, with no repition of the notes at either extreme (see Figure 2 for a C-major arrangement). Notice that the initial processor was not placed at the beginning of the scale. In fact, we found it more effective for the tune to emerge at some different point in the ring, rather than being tied to the initial processor.

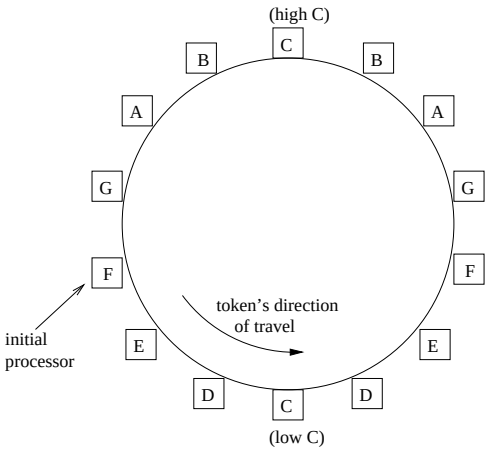

Figure 2: Arrangement for Binary State Token Ring

With every processor beginning in state 0, the algorithm was initiated. The girls played their note, changed their state, and observed the progress of the virtual token around the ring. One nice feature of using a major scale is that every note has the same duration, so the tune (simple as it is) is virtually guaranteed to emerge, despite the participants not being given any timing information. They were told nothing beyond: "when you have the token, play the note, and copy your neighbor's value". (This rule was modified appropriately for the initial processor and an assistant stayed next to her to be sure she correctly understood her modified rule.)

Another nice feature of the ascending/descending scale is that it sounds like a continuous melody, without beginning or end. This reinforces the notion that the token circulates around the ring continuously.

After the participants were comfortable with how to execute the algorithm, a fault was introduced. The state of one processor was modified (by an assistant acting as a "fault demon"). For simplicity, this fault was not introduced in the vicinity of the original token. The effect of such a fault is to introduce *extra* tokens in the system. In particular, the neighbor of the faulty processor will now have a token and will propagate that token around the ring. In addition, the faulty processor itself will have a token, and so will propagate that token too around the ring. The result is three separate waves of tokens circulating around the ring, creating a cacophony and dramatically illustrating the loss of mutual exclusion.

[1] A more accurate description of this variable is an encoding of how many times the processor has held the token, modulo 2.

This algorithm does not recover from a fault. Multiple tokens will continue to circulate around the ring. Students saw how there was always at least one token in the ring (no configuration results in silence). But they also saw how the algorithm might never recover from the insertion of extra tokens.

5.3 k-State Token Ring.

The difficulty with the binary state token ring is that the initial processor cannot distinguish between a legitimate wave and one that was initiated through a fault. With only two states to distinguish token waves, three (or any odd number) of waves might circulate indefinitely. The solution is to use more than two states. The k-state algorithm uses the same number of states as there are processors in the ring (in our case, 14). Thus, each wave has a different number and can be distinguished. The initial processor will only propagate a wave with the same number as the one it initiated last.

The algorithmic rule for each process is exactly the same as before: If my left neighbor's state is different than mine, I use the resource then copy their state. For the initial process, if its left neighbor's state is the same, it uses the resource then increments by one.

Again the volunteers were arranged in a circle and given a page for indicating current state. This time, however, possible state values ranged from 0 to 13 inclusive. Also, this time a different arrangement of tone bars was used. Thus, the students did not know what tune they would be playing. The algorithm would have to stabilize for the melody to become clear.

The tune used was *Twinkle Twinkle Little Star* (C-C-G-G-A-A-G-F-F-E-E-D-D-C). It has the advantages of being easily recognizable, having notes of mostly equal duration, and having 14 notes. Another nice feature of this tune is that no one note occurs more than 3 times. Thus, 3 copies of a single octave of tone bars suffices to assemble the melody.[2] Again, the tune was *not* aligned to begin with the initial processor.

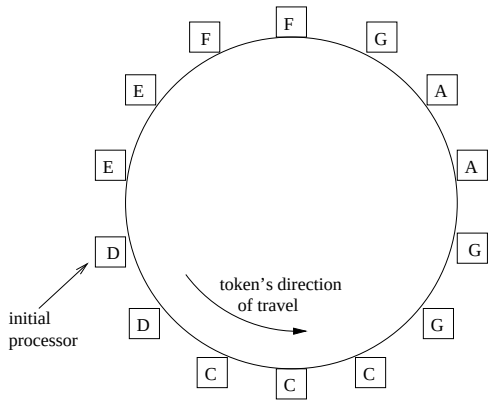

Figure 3: Arrangement for k-State Token Ring

[2]We also considered *Frère Jaques* and The Ohio State University's alma mater, *Carmen Ohio*. Both are 14-note tunes with relatively equal duration notes. However, the former requires more than three instances of an individual note and the later's recognition is less universal.

With everyone in the same initial state (all processors in state 0) the algorithm was initiated. Before the tune could become discernible, however, faults (modifications to state) were introduced. Like before, these modifications result in extra tokens being added to the system. With multiple notes being played concurrently, any semblance of a melody was lost.

The self-stabilizing nature of this algorithm, however, meant that gradually the extra tokens were removed. Gradually, the system returned to a single token and, to the students' surprise and delight, the melody of *Twinkle Twinkle Little Star* emerged.

As further evidence of the self-stabilizing nature of this algorithm, it was initiated in a random state (each student picking a number between 0 and 13 inclusive). When the algorithm ran, there was musical chaos with practically all the notes being played and many tokens being propagated. Even from this extremely chaotic configuration, however, the tune eventually emerged.

Students observed first-hand the role of the initial processor in removing extra tokens. Because "using the resource" meant playing a note, there was immediate feedback when the safety property was violated (after a fault) and immediate gratification when the algorithm recovered and a tune emerged.

The fault tolerance exercise requires one assistant to supervise the initial processor and another assitant to introduce faults. This activity required 45 minutes.

6 Feedback From Participants

The entire workshop ran for a total of 5 days, 9 am to 5 pm, plus one evening (a total of 44 hours). The CS module described in this paper ran for 2.5 hours during the morning of the fourth day.

At the end of each day, students were given feedback forms with which to evaluate the day's activities. The day of the CS module, their feedback form included the following quantitative questions.

How much did you know about computer science before doing the activities? On a scale of 1 (none) to 5 (a lot), the average response was 2.8, with a standard deviation of 1.2

Having done the activities, does computer science seem more or less interesting than before? On a scale of 1 (much less interesting) to 5 (much more interesting), the average response was 4.0, with a standard deviation of 0.8.

Each activity was also rated on a scale of 1 (low) to 5 (high). All the activities were well received, with the fault tolerance activity being rated the highest. This feedback is summarized in Table 1.

Activity	Average	Std. Dev.
Robot Programming	4.4	0.8
Parallel Programming	4.4	0.7
Fault Tolerance	4.6	0.7

Table 1: Participant Evaluation of Workshop Modules

In addition to these quantitative questions, students were also asked: "**What is the most important thing you feel you learned about computer science?**" Responses fell in four categories. Three students wrote something similar to, "It's really fun!" Six students commented on programming and software engineering. Comments included: "How to write programs", and "Computers need specific instructions to work right." Seven students mentioned parallel programming. Comments included: "The more 'chefs' you use, the faster will the program finish", and "Sequential programs are slow." Finally, five students mentioned fault tolerance. Comments included: "I learned that computers have 'faults' that can be fixed!", and "how to make a program recover itself."

At the end of the week, students were given a survey to evaluate the workshop as a whole. When asked to evaluate presentations on a scale of 1 (low) to 5 (high), students gave the computer science module an average grade of 4.6, the highest of all presentations.

Students were also asked: "**In which activity did you learn the most about engineering?**" There were 15 different units introducing various science and engineering disciplines (including chemistry, civil engineering, mechanical engineering, aeronautical engineering, environmental engineering, astronomy, industrial engineering, *etc.*). Of these, the computer science module had the highest number of responses (8).

It should be noted that this exit survey was completed on the last day of the workshop, the same day that students had participated in the "highlight" activity of building a hovercraft. Building (and riding!) these hovercrafts was extremely entertaining and was greatly enjoyed by the participants. Nevertheless, when asked in which activity they *learned* the most, more students selected the computer science module from an earlier day.

7 Conclusions

Traditional approaches to introducing computer science concepts at a middle school level have often focussed on simple imperative programming environments. We developed a 2.5 hour module to introduce middle school girls to advanced algorithmic concepts in computer science. The material for this module (slides, handouts, and code) is available at URL `http://www.cis.ohio-state.edu/~paolo/FESC02`.

The role of musical tunes to illustrate the token ring algorithms can not be overstated. The musical element made it immediately apparent when a fault had occurred. It also created a sense of gratification and satisfaction when the algorithm stabilized and a melody emerged. We expect this strategy for anthropomorphizing distributed algorithms could be used for a variety of algorithms. There are many variations on stabilizing token rings (*e.g.*, the 4-state ring, and 3-state ring) to which the technique could be directly applied. Other advanced problems, such as distributed consensus, might also be amenible to similar treatment.

8 Acknowledgements

The authors gratefully acknowledge the help of Sandip Bapat who assisted in the implementation and tuning of the robot programming activity. The xylophone tone bars were borrowed from Amy Giles in the university's School of Music (Music Education). Thanks also to the members of the Distributed Components research group at The Ohio State University who helped test and run these activities, in particular: Florina Comanescu, Scott Pike, Nigamanth Sridhar, and Hilary Stock. Finally, thanks to Linda Weavers for spearheading the week-long workshop and inviting our participation.

This work was supported by NSF ITR grant CCR-0081596, and an Ameritech Faculty Fellowship. The workshop was funded by an NSF PECASE award.

References

[1] Camp, T. The incredible shrinking pipeline. *Communications of the ACM 40*, 10 (October 1997), 103–110.

[2] Countryman, J., Feldman, A., Kekelis, L., and Spertus, E. Developing a hardware and programming curriculum for middle school girls. *SIGCSE Bulletin 34*, 2 (June 2002), 44–47.

[3] Dijkstra, E. W. Self stabilizing systems in spite of distributed control. *Communications of the ACM 17*, 11 (November 1974), 643–644.

[4] Harvey, B. *Computer Science Logo Style*, 2nd ed. MIT Press, February 1997.

[5] Rifkin, A. Teaching parallel programming and software engineering concepts to high school students. In *Proceedings of the 25th SIGCSE Symposium on Computer Science Education* (March 1994), ACM, pp. 26–30.