Session 1520

An Invariant Pattern-based Approach to Develop Concurrent Programs

M. Mizuno[1], G. Singh[1], M.L. Neilsen[1], D.H. Lenhert[2], N. Zhang[3], and A.B. Gross[4]

[1] Department of Computing and Information Sciences, Kansas State University (KSU) {masaaki,singh,neilsen}@cis.ksu.edu*
[2] Department of Electrical and Computer Engineering, KSU, lenhert@ksu.edu*
[3] Department of Biological and Agricultural Engineering, KSU, zhangn@ksu.edu*
[4] The IDEA Center, 211 S. Seth Child Road, Manhattan, Kansas, agross@ksu.edu*

Abstract

In recent years, the importance of concurrent programming has increased. However, many programmers are not appropriately trained to write correct and efficient concurrent programs. The techniques that most Operating Systems (OS) textbooks teach are ad-hoc, and such ad-hoc techniques are far too error-prone for solving complex synchronization problems. The global invariant approach developed by G. Andrews is much more formal and structured, and we have been teaching this approach since 1992 at Kansas State University. One possible drawback of the invariant approach is the difficulty to identify an appropriate invariant for a given synchronization requirement. To cope with this problem, we have developed a set of useful synchronization patterns and their solution invariants. Using the patterns, we can solve a wide-variety of synchronization problems found in many advanced OS textbooks. In Fall 2001, we successfully taught our pattern-based approach in our graduate-level OS course. In this paper, we will present our methodology and report qualitative and quantitative evaluation of the methodology by students in the classroom setting.

1 Introduction

In recent years, concurrent programming has become the norm rather than the exception in many applications. The advantage of concurrent programming is that an individual thread (or process) is written as a sequential program focusing only on its sequential activities, and the coordination among activities by different threads is localized to a small amount of *synchronization code*.

* This work was supported in part by the National Science Foundation under NSF-CRCD Grant #9980321 and DARPA Order K203/AFRL #F33615-00-C-3044.

Copyright © 2002, American Society for Engineering Education

However, informal conversations with system designers/engineers at Boeing and 3COM on separate occasions have revealed that in industry, many programs are written in the form of a sequential program or a simple non-preemptive event-driven program. Even in cases where multiple threads are used, programmers tend to be conservative in the use of complex synchronization code, and they often adopt the simple but inefficient "lock everything" approach to avoid potential race conditions and deadlocks. As the complexity of software systems grows, such simple strategies will become inadequate and inappropriate.

The design practice to use simple software structures can be attributed to the following:

1. Due to long life cycles of commercial systems, many programs use legacy code originally developed for much simpler systems. However, these programs are becoming increasingly difficult to maintain and satisfy all system requirements as the complexity of the systems grow.
2. Many programmers and designers are not appropriately trained to write correct and efficient concurrent programs. Most Operating Systems (OS) textbooks explain synchronization by showing solutions in terms of low-level synchronization primitives for some well-known synchronization problems, such as the readers/writers problem and the dining philosophers problem. Such solutions do not generalize to complex real-life synchronization problems in various primitives. Furthermore, this approach only teaches ad-hoc techniques for development of synchronization code, and as stated in [1], ad-hoc techniques are far too error-prone for solving complex synchronization problems. Since bugs caused by erroneous synchronization code are difficult to detect and debug, there is reluctance among the programmers to develop sophisticated concurrent programs.

We have focused our research on the second issue. Through the support of an NSF Combined Research and Curriculum Development grant, we have developed a formal methodology to construct reliable concurrent programs [8,10] and an interdisciplinary curriculum for teaching the methodology in the context of embedded systems development [11]. Our methodology is based on G. Andrews' global invariant approach [1,2,3] in which a programmer specifies synchronization using formulas in formal logic in terms of an invariant, and mechanically translates the invariant to low-level synchronization code in various languages and primitives. Since the translations are guaranteed to preserve the invariant, the methodology yields correct solutions in terms of the synchronization specification. We have been teaching the global invariant approach in our graduate-level Advanced Operating Systems course (CIS720) since 1992.

The global invariant approach has many advantages. First, it is a formal approach that enables verification and synthesis of programs being developed. Second, the most important activity in the programming process lies at a high level; namely, specifying global invariants. Once an appropriate global invariant is specified, much of the rest of the process is mechanical. Furthermore, global invariants are platform (synchronization primitive) independent. For example, if the platform is switched from a semaphore-based to a monitor-based system, we only need to apply an appropriate translation to yield a monitor-based program.

Copyright © 2002, American Society for Engineering Education

One possible drawback of the global invariant approach is the difficulty to identify an appropriate global invariant that correctly and accurately captures the synchronization requirement. Interestingly, writing low level code (using an ad-hoc approach) seems to be easier for many students than specifying the formal safety property in terms of an invariant. Many students who took CIS720 prior to 2001 (that is, before we developed the pattern-based approach) remarked that they had developed code in low-level primitives first by trial and error and then obtained a global invariant by observing the code.

To cope with the difficulty of identifying global invariants, we have developed a set of useful synchronization patterns and their solution global invariants [8,10]. These patterns work as basic building blocks, and their solution invariants can be composed to produce global invariants for more complex synchronization. Most problems in OS textbooks can be solved easily by composing these synchronization patterns. We are currently trying to apply our pattern-based approach to real-life synchronization problems found in military applications through the support of a DARPA grant.

In Fall 2001, we taught our pattern-based approach for the first time in a formal class-room setting in CIS720. In this paper, we will present our methodology and report qualitative and quantitative evaluation of the methodology by students in the course.

2 Overview of our methodology

Our pattern-based approach [8,10] is developed in the framework of aspect-oriented programming [7], in which properties that must be implemented are classified into the following two types:

- *Components* that can be captured in general procedures, and
- *Aspects* that cannot be clearly encapsulated in general procedures, such as synchronization.

We apply a well-regarded Object-Oriented methodology, called Rational Unified Process (RUP) [6], to develop component (sequential) code. One of the key processes in RUP is to describe the sequential behavior of each component in a *scenario* (also called a *use-case realization*). Component code can be effectively obtained from a scenario, if the scenario is described in enough detail. Each component is executed by one or more threads. Therefore, a system consisting of multiple components is executed potentially by multiple threads. Therefore, in such a system, we need to implement appropriate synchronization to control the behavior of those threads.

G. Andrews developed a formal approach (called the *global invariant approach*) to develop synchronization code [1,2,3]. In this approach, for a given problem, we first specify a *global invariant* that implies the safety property[1]. Then, the programmer derives a so-called *coarse-grained solution* from the global invariant. The process of deriving a coarse-grained solution is based on *Programming Logic* and is strictly mechanical. The resulting coarse-grained solution preserves the global invariant. Next, the coarse-grained solution is mechanically translated to *fine-grained* synchronization code. The translation preserves the global invariant; therefore, the

[1] The safety property asserts that the program never enters a bad state.

Copyright © 2002, American Society for Engineering Education

resulting program is guaranteed to satisfy the safety property of the synchronization requirement. Many translations from a coarse-grained solution to fine-grained synchronization code exist, including translations to semaphore code, monitor code, active monitor code in message passing systems[1], and Java synchronized blocks[9].

Our methodology uses RUP to develop sequential component code and the global invariant approach to develop synchronization code [8,10]. We will explain our methodology by using the following problem, called the *Roller Coaster Ride Problem*, found in advanced concurrent programming textbooks[1,5].

> *There are n passengers and one roller coaster car. The passengers repeatedly wait to take rides in the car, which holds C passengers. The car can go around the track only when it is full. The car takes the same T seconds to go around the track each time it fills up. After getting a ride, each passenger wanders around the amusement park for a random amount of time before returning to the roller coaster for another ride.*

The first step of our methodology is to apply RUP to develop scenarios.

The scenario for the car thread is (it repeats the following steps):
[C1] {Assertion: the car is empty} *wait until C* passengers have gotten on the car
[C2] {Assertion: *C* passengers are on the car} go around the track (elapse *T* seconds)
[C3] stop and have the passengers get off the car
[C4] *wait until* all *C* passengers have left

The scenario for a passenger thread is (it repeats the following steps):
[P1] *wait until* his turn comes and get on the car
[P2] {Assertion: the passenger is on the car} *wait until* the car goes around and stops
[P3] {Assertion: the car has stopped} get off the car
[P4] wander around the amusement park (elapse a random amount of time)

Note that one thread executes the sequential component code obtained from the passenger scenario and that *n* concurrent threads execute the sequential component code obtained from the passenger scenario.

The second step is to identify *synchronization regions* and *clusters* in the scenarios. Synchronization regions are segments in scenarios in which the execution must wait (be blocked) until some condition holds or in which the execution changes some condition which is waited for by other threads. These synchronization regions are divided into partitions called *clusters*, based on the reference relations. In the above example, steps C1, C3, and C4 constitute synchronization regions, denoted R_{C1}, R_{C3}, and R_{C4}, respectively. Steps P1, P2, and P3 constitute synchronization regions, denoted R_{P1}, R_{P2}, and R_{P3}, respectively. Since the execution in region R_{C1} waits for condition that could be changed to true by the execution in region R_{P1}, they form a cluster, denoted by (R_{C1}, R_{P1}). Similarly, (R_{C3}, R_{P2}) and (R_{C4}, R_{P3}) are clusters. Refer to Figure 1.

Copyright © 2002, American Society for Engineering Education

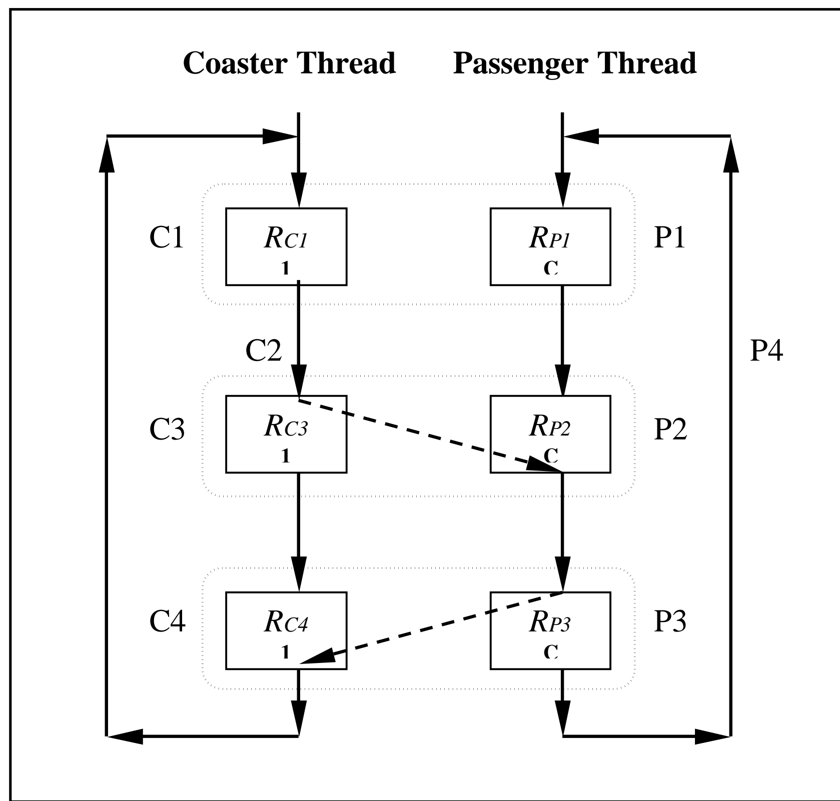

Figure 1. Roller Coaster Problem

The third step is to identify an appropriate global invariant for each cluster. The global invariant for the cluster implies the safety property for the cluster. Most synchronization requirements found in exercise problems in OS textbooks are covered by our synchronization patterns. Therefore, programmers merely need to identify appropriate patterns for the clusters. For example, the synchronization requirement of cluster (R_{C1}, R_{P1}) is that one thread in R_{C1} and C threads in R_{P1} wait for each other's arrival at their respective regions and leave their regions at the same time. This requirement is captured by synchronization pattern called **Barrier**. After **Barrier**, the thread leaving R_{C1} is sure that C threads have arrived at R_{P1}. Similarly, each of the C threads leaving R_{P1} is sure that one thread has arrived at R_{C1}. In the scenarios given in the first step, such informal assertions are added after the synchronization regions in {…} to give readers the feel for the safety properties. The synchronization requirement of cluster (R_{C3}, R_{P2}) is that C threads in region R_{P2} wait for arrival of one thread at region R_{C3}. This requirement is captured by pattern called **AsymBarrier** (Asymmetric Barrier). Similarly, the synchronization requirement of cluster (R_{C4}, R_{P3}) is captured by **AsymBarrier**. For each synchronization pattern, we provide a solution invariant. Therefore, this step, which is the most challenging step and the core process in the global invariant approach, is greatly simplified.

Copyright © 2002, American Society for Engineering Education

The fourth step is to develop a coarse-grained solution and fine-grained synchronization code for each invariant and weave the resulting synchronization code to the component sequential code developed from the scenarios. This step is purely mechanical and can be automated. We have developed supporting tools that (1) obtain a coarse-grained solution from a global invariant and that (2) translate a coarse-grained solution to Java code [4]. Other researchers have also developed similar tools [12].

3 Pattern-based approach

In this section, in order to give readers the feel for the patterns and their applications, we will present a few synchronization patterns and show two applications of the patterns. Since this is not a technical paper specifically targeting researchers of concurrent programming, we will not elaborate on the complete set of patterns. In fact, as we encounter challenging problems that cannot be solved using the current set of patterns, we add new patterns to the set; therefore, the set of patterns is still expanding as of today. Furthermore, we will not present solution invariants. Instead, only high-level description of the patterns will be given. This high-level description, without concern about the technical and formal details of the invariants, is what ordinary programmers need to understand in order to develop concurrent programs using our pattern-based approach. Our tools hide invariants and the translation process from programmers. For interested readers, technical details of invariants and complete solution code for a large set of synchronization problems can be found elsewhere [8].

3.1 Example synchronization patterns

- **Bound**(R,n): A cluster consists of a single synchronization region R. The synchronization specification is that at most n threads can be in region R at any point in time. For example, if a thread comes to enter R while there are already n threads in R, the thread cannot enter and must wait until one thread leaves R.
- **Exclusion**($R_1,R_2,\ldots,R_n$): A cluster consists of n synchronization regions, $R_1,R_2,\ldots,R_n$. At any point in time, threads can be in at most one synchronization region out of the n regions. For example, if a thread comes to enter R_1 while there are threads in R_2, the thread cannot enter and must wait until all the threads in R_2 leave.
- ***k*-MuTex**($R_1,R_2,\ldots,R_n$, k) (k-Mutual Exclusion): A cluster consists of n synchronization regions, $R_1,R_2,\ldots,R_n$. At any point in time, at most k threads can be in (any) regions in the cluster.
- **Barrier**((R_1,N_1), (R_2,N_2),…, (R_n,N_n)): A cluster consists of n synchronization regions, $R_1,R_2,\ldots,R_n$. N_i threads entering R_i for $i = 1,\ldots, n$ meet, form a group, and leave the respective regions together. Recall that in the Roller Coaster Problem, we used **Barrier**(($R_{C1},1$),(R_{P1},C)), in which the car thread in R_{C1} and C passenger threads in R_{P1} needed to meet, form a group, and leave together.

Copyright © 2002, American Society for Engineering Education

As variations of the **Barrier** pattern, we have:

- **AsymBarrier**([(R_{S1},N_{S1}), (R_{S2},N_{S2}),…, (R_{Sn},N_{Sn})], [(R_{W1},N_{W1}), (R_{W2},N_{W2}),…, (R_{Wm},N_{Wm})]) (Asymmetric Barrier), in which N_{Si} threads entering regions R_{Si} for $i = 1,\ldots, n$, triggers the departure of N_{Wj} threads from regions R_{Wj} for $i = 1,\ldots,m$. Recall that in the Roller Coaster problem, we used **AsymBarrier**([(R_{C3},*1*)],[(R_{P2},*C*)]), in which the arrival of the car thread at R_{C3} triggered the departures (getting out of the car) of *C* passenger threads from R_{P2}. Similarly, we used **AsymBarrier**([(R_{P3},*C*)], [(R_{C4},*1*)]), in which the arrivals (completion of getting out of the car) of *C* passenger threads triggered the departure of the car thread from R_{C4}.
- **Barrier with Information Exchange**((R_1,N_1), (R_2,N_2),…, (R_n,N_n)): in which N_i threads in regions R_i, for $i = 1,\ldots, n,$ forming a group exchange information before leaving together.

3.2 Applications of patterns

As the names indicate, some patterns directly solve common synchronization problems, such as the critical section problem (***k*-MuTex** (mutual exclusion) with *k* equal to 1) and a simple barrier synchronization with two threads (**Barrier**((R_1,*1*),(R_2,*1*))). We will show one example that is solved by composition of the invariants of two patterns and another example that is an extension of the Roller Coaster problem.

3.1.1 Readers/Writers problem

The problem description is

> *There is a single buffer that is accessed by multiple reader threads and writer threads. Reader threads and writer threads cannot access the buffer at the same time. Any number of reader threads can access the buffer concurrently, but at most one writer thread can access the buffer at any point in time.*

Suppose that reader and writer threads access the buffer in synchronization regions R_R and R_W, respectively. The invariant for the cluster formed by R_R and R_W is given by composition of the invariants of two patterns as **Exclusion**(R_R, R_W) **&&** **Bound**(R_W,*1*), where **&&** is a logical conjunction operator.

3.3.1 Multiple Coaster Cars problem

The problem is to extend the Roller Coaster Ride Problem to have *M* cars.

If we try to solve this problem directly using low-level primitives (using an ad-hoc approach), the solution would be quite different from that for the single coaster problem. However, using our pattern-based approach, the solution for the multiple coaster problem is obtained only by changing the invariant for cluster (R_{C1},R_{P1}) from **Barrier** to **Barrier with Information Exchange**. The information to be exchanged in the cluster is the handle (or identifier) of the car thread. Later, passengers synchronize at clusters (R_{C3}, R_{P2}) and (R_{C4}, R_{P3}) with the particular car identified by the handle.

Copyright © 2002, American Society for Engineering Education

4 Evaluation

4.1 Exams

In CIS720 offered in Fall 2001, we covered the following topics:

1. Review of processes, threads, and development of concurrent programs in a traditional manner (called an ad-hoc approach in this paper),
2. Programming Logic and the global invariant approach to develop synchronization code,
3. Development of Object-Oriented sequential programs using RUP,
4. Our methodology to develop concurrent programs (discussed in Section 2), which combines RUP to develop sequential component code and the global invariant approach to develop synchronization code,
5. The synchronization patterns and their solution invariants.

We gave three exams. The first exam covered the above 1 and 2. The exam included questions about (1) development of fairly simple synchronization code in monitors using an ad-hoc approach, (2) verification of simple sequential programs using Programming Logic, (3) identification of appropriate global invariants for two synchronization requirements; one being simple and another being reasonably challenging, and (4) translation of a global invariant to semaphore code. The second exam covered the above 3, and the third exam covered the above 4 and 5. The third exam consists of questions to develop synchronization code in active monitor and Java for two challenging problems using patterns. On of these questions would not have been given in exams prior to 2001 (that is, before we developed the pattern-based approach), because it is too difficult.

We always attempt to set the difficulty of questions such that the expected average score would be around 60 out of 100, and most of the time, average scores resulted in that score range. This time, the average scores of the first and the third exams were 62.1 and 87.2, respectively, in the class of 35 students. Since questions were different and the maturity of the students on concurrent programming differed in the first and the third exams, it is difficult to analyze only from the average scores. However, from the fact that 86% of the students, some of whom received very low scores in the first exam, obtained scores over 80 and 46% obtained over 90 in the third exam, we conclude that our pattern-based approach is very effective to teach concurrent programming.

4.2 Students' evaluation

We conducted a survey on the subjects covered in the course, including the following questions:

Give scores between 1 and 5 to the following methodologies in developing synchronization code in terms of the ease of use (1 is the most difficult and 5 is the easiest).

Q1. ad-hoc approach
Q2. global invariant approach without using patterns
Q3. global invariant approach with patterns

Copyright © 2002, American Society for Engineering Education

Note that the survey was conducted before the third exam; that is, before many students knew that they would have been able to solve challenging synchronization problems using patterns in the third exam.

25 students responded to the survey. The distribution and average score of each question are given in Table 1.

	1 (most difficult)	2	3	4	5 (easiest)	Average
1. Ad-hoc Approach	6	7	10	2	0	2.3
2. Global invariant approach without using patterns	1	2	8	11	2	3.5
3. Global invariant approach with patterns	0	1	3	7	13	4.3

Table 1. Frequencies and average of students' response

Eleven students wrote comments, all positive about the pattern-based approach. Some of the comments were:

- Pattern-based approach is very nice. Using patterns, the global invariant approach, and RUP, I am confident that I can write many of the complex concurrent programs. Pattern-based approach has good features that I haven't seen in any approach in developing concurrent programs.
- It is absolutely wonderful approach. Specially [sic] the use of patterns [and] translations make concurrent programming really easy and very interesting.
- This is the best method I have come to know to develop concurrent programs.
- I really liked the pattern-based GI, RUP approach. It has made it easy to write concurrent programming. …. Pattern-based approach is really interesting and helpful.
- It is an excellent approach and makes the programmers life a lot easier!! Ad-hoc approach involved too much work.

5 Conclusion

Recent years, the importance of concurrent programming has increased. However, there is reluctance among programmers to develop concurrent programs since bugs caused by erroneous synchronization code are difficult to detect and debug. In fact, most textbooks that teach concurrent programming introduce synchronization by showing solutions in terms of low-level primitives for some well-known synchronization problems. This approach would teach only ad-hoc techniques for development of synchronization code, and as stated in [1], ad-hoc techniques are far too error-prone for solving complex synchronization problems.

The global invariant approach developed by G. Andrews is much more formal and structured, and yields solutions that are "correct by construction" [1,2,3]. One possible drawback of the global invariant approach is the difficulty to identify an appropriate global invariant that correctly and accurately implies the safety property of a given synchronization requirement. To

Copyright © 2002, American Society for Engineering Education

cope with this problem, we have developed a set of useful synchronization patterns and their solution global invariants [8,10]. We have successfully introduced the pattern-based approach in our advanced Operating Systems course in Fall 2001. Most of the students could solve very challenging synchronization problems using patterns.

We are currently trying to apply our pattern-based approach to real-life synchronization problems found in military applications through the support of a DARPA grant.

References

[1] G.R. Andrews, "Concurrent Programming, Principles and Practice", Benjamin/Cummings Publishing Co., 1991.

[2] G.R., Andrews, "Foundations of Multithreaded, Parallel, and Distributed Programming", Addison Wesley, 2000.

[3] A.J. Bernstein and P.M. Lewis, "Distributed Operating Systems and Algorithms", Jones and Bartlett, 1993.

[4] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno, "Invariant-based specification, synthesis, and verification of synchronization in concurrent programs", will appear in the Proceedings of International Conference on Software Engineering (ICSE), 2002.

[5] S.J. Hartley, "Concurrent Programming – The Java Programming Language", Oxford University Press, 1998.

[6] I. Jacobson, G. Booch, and J. Rumbaugh, "The Unified Software Development Process", Addison Wesley, 1999.

[7] G. Kiczales, J. Lampoing, A. Mendhekar, C. Maeda, C. Lopez, J. Loingtier, and J. Irwin, "Aspect-oriented programming", In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, 1997.

[8] M. Mizuno, "A pattern-based approach to develop concurrent programs – Part 1/Part2", Kansas State University Technical Report, 2001-02/03 (under review for publication).

[9] M. Mizuno, "A structured approach for developing concurrent programs in Java", Information Processing Letters, Vol. 69, No 5, pp.232-238, 1999

[10] M. Mizuno, M.L. Neilsen, and G. Singh, "A structured approach to develop concurrent programs in UML", In Proceedings of the International Conference on the Unified Modeling Language (UML 2000), pp.451-565, Oct 1-3, 2000.

[11] M.L. Neilsen, D.H. Lenhert, M. Mizuno, G. Singh, N. Zhang, and A.B. Gross, "An interdisciplinary curriculum on real-time embedded systems", 2002 ASEE Annual Conference, Session 1526

[12] T. Yavus-Kahveci, and T. Bultan, "Specification, Verification, and Synthesis of Concurrency Control Components", will appear in the Proceedings of International Conference on Software Engineering (ICSE), 2002.

Copyright © 2002, American Society for Engineering Education

Biographical Information

MASAAKI MIZUNO is a Professor in the Department of Computing and Information Sciences at Kansas State University. His research interests include operating systems and distributed systems.

GURDIP SINGH is an Associate Professor in the Department of Computing and Information Sciences at Kansas State University. His research interests include network protocols, distributed systems, and verification.

MITCHELL L. NEILSEN is an Assistant Professor in the Department of Computing and Information Sciences at Kansas State University. His research interests include real-time embedded systems, distributed systems, and distributed scientific computing.

DONALD H. LENHERT is the Paslay Professor in the Department of Electrical and Computer Engineering at Kansas State University. His research interests include embedded systems and digital testing.

NAIQIAN ZHANG is a Professor in the Department of Biological and Agricultural Engineering. His research interests include sensors and controls for biological and agricultural systems.

AMY B. GROSS, Associate Director of The IDEA Center, served as the external evaluator for the NSF-CRCD grant. The IDEA Center's mission is to assist colleges and universities assess and improve teaching, learning, and administrative performance.

Copyright © 2002, American Society for Engineering Education