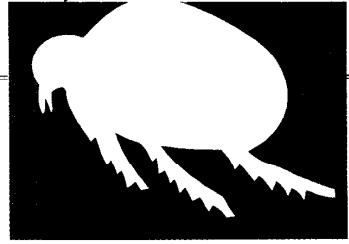


A General Framework for Debugging

KEIJIRO ARAKI, ZENGO FURUKAWA, *and* JINGDE CHENG, *Kyushu University*

◆ *Programmers have no clear idea how to systematically debug a program, which makes it difficult to share and evaluate methods. This article presents the requirements for a general debugging framework.*

Debugging is said to be the least established area in software development: Industrial developers have no clear ideas about general debugging methods or effective and smart debugging tools, yet they debug programs anyway.

Debugging requires much experience in program development because it relies on heuristic insights. Because it is not established as a disciplined task, it is difficult to share as a systematic method. In an effort to correct this, we have developed a process model that establishes a minimum set of requirements for systematic debugging.

Our process model views debugging as an iterated process of developing hypotheses and verifying or refuting them. Hypotheses may concern the location of bugs, the causes of errors, expected program behavior, and how to modify the program to correct errors, among other things.

STATE OF THE ART

The most primitive way to debug a program is to insert so-called debug statements, which during execution print information like what statements are executed or what values particular variables have at a particular point. Debug statements let you observe a program's behavior.

The most important considerations in this method of debugging are where to insert debug statements, what kind of information to print, and what to comprehend by analyzing the output.

Another approach is to use debugging tools, many of which provide only primitive, low-level facilities. These tools help programmers insert breakpoints, inspect the states at those points during program execution, change the values of variables and program parts, and continue the execution.

0740-7459/91/0500/0014/$01.00 © IEEE

Other tools include tracers, which let you observe or analyze a program's execution history, and symbolic debuggers, which provide high-level interfaces.

The same considerations hold when using debugging tools that hold when using debug statements — what to observe and what information will be obtained. Some programmers do not completely trust the information obtained with debuggers, insisting that debuggers sometimes report incorrect information.

In addition to this concern, debugging tools are not yet effective because they do not provide enough abstraction to represent and retrieve information at the specification and computation-model level. Researchers have tried different ways to manage the huge amount of output from debuggers and have used graphics and animation to try to make program execution more understandable, but so far these attempts have not produced debuggers that are effective enough for high-level debugging.[1]

Concurrent programs present an even more difficult debugging problem because it is difficult to repeat the execution that outputs an error.[2,3] When an error is found in a concurrent program, the very first thing programmers try to do is reproduce the error. They may spend a great deal of time and effort only to find the conditions that caused the error by chance. Programmers often say that debugging a concurrent program is almost complete when you've figured out how to reproduce the error.

Another serious problem overall is the effect the probe or monitoring device itself has on program behavior during execution. It's possible that the probes themselves may give programmers incorrect information about a program's behavior.

With these problems in mind, is it possible to develop a general framework for debugging that programmers can follow? We believe so. As we describe here, you can derive many processes from our general model, from traditional debugging processes to an algorithmic process for a class of logic programs[4] to an execution monitoring tool for concurrent Ada programs.[5]

You can derive many processes from our general model, from traditional debugging processes to an algorithmic process for a class of logic programs to an execution monitoring tool for concurrent Ada programs.

DEBUGGING PROCESS

In this article, "error" means the difference between a program and its specification — the difference between the behavior requested by the specification and the behavior performed by the program — and "bug" means the cause of an error. We assume that the specification has no error, that the cause of the error — the bug — is in the program. We also assume that errors are detected either when developers test a program or when users use it, and we assume these errors are to be corrected by debugging.

Debugging is the process of locating and correcting errors in a program in which errors have been detected. In locating the errors and grasping their causes, programmers develop hypotheses about the errors and their causes, and verify or refute these hypotheses by examining the program. In correcting the errors, programmers again develop hypotheses about how to modify the program and verify or refute them.

Figure 1 shows our debugging process model: Programmers begin with a set of hypotheses, modify that set, select hypotheses for verification, and verify or refute the selected hypotheses until the bug is fixed.

Hypothesis set. In our model, hypotheses relate to source code, its specification, and its behavior. In fact, programmers hypothesize about the errors in the program, including the places in the program where errors are supposed to occur, the supposed causes of the errors, behaviors that are supposed to be correct or incorrect, and modifications that are supposed to correct the errors.

These hypotheses include the facts that have been proved so far about the properties of the program and its errors, although in this article we also call these facts hypotheses. The programmer's empirical knowledge of development, the program itself, and its specifications are also included in the hypothesis set. For simplicity, our debugging process model starts with error reports as the initial hypothesis set.

Hypothesis-set modification. As debugging proceeds, programmers will modify the hypothesis set through generation, refinement, and authentication.

♦ Generation. You create new hypotheses from known facts and existing hypotheses. To generate qualified hypotheses, you need much development experience and good insight into the application domains.

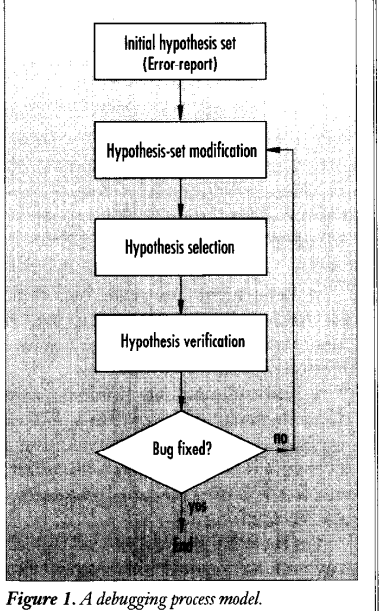
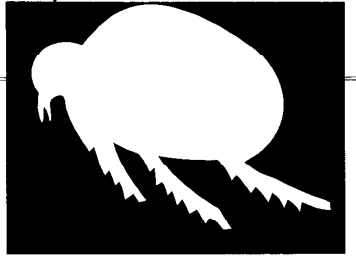


Figure 1. A debugging process model.



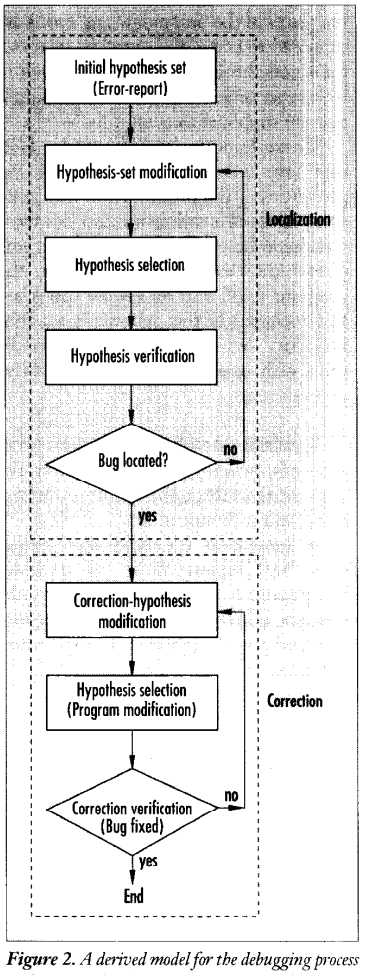


Figure 2. A derived model for the debugging process with separate localization and correction phases.

♦ Refinement. You refine hypotheses by making them more constrained or more detailed according to the results of previous hypothesis verification.

♦ Authentication. Verification changes the authenticity of a hypothesis. For example, if you prove that a hypothesis is at all true, it is no longer a hypothesis but a true fact, and its conflicting hypotheses, if any, become untrue.

On the other hand, if you fail to verify a hypothesis, you must not remove it from the set but replace it with its refutation. Even if you fail to verify hypotheses about the errors and cannot realize their true characteristics, the effort of debugging has been meaningful. An accumulation of refuted hypotheses gives some information about the errors and contributes to the discovery and removal of bugs.

Hypothesis selection. Randomly selecting a hypothesis to verify is not efficient. Instead, you should select a hypothesis according to a strategy, specified with tactics such as:

♦ Simplify the error condition. To find the causes of an error, you try to find simpler conditions that cause the same error.

♦ Narrow the suspicious region. To locate the bug, you try to narrow the region where an error occurs, beginning with, for example, the entire program, then a module, then a statement, and so on.

♦ View selection. To examine a program from a particular point of view, you select a hypothesis that describes the program according to the particular view. For example, to debug a concurrent program, you pay attention to the interaction between concurrent processes.

♦ Expand the certified region. When you correct an error by modifying the program, you must verify that the modification removes the bug and creates no new errors. In contrast with narrowing the suspicious region, you examine the modification's effect from smaller to larger regions, and finally to the whole program.

♦ Weighting. When you have more than one candidate hypotheses, you select the most significant one for verification. To this end, you need a way to weigh the significance of each hypothesis.

Hypothesis verification. Programmers use four techniques to verify a hypothesis by examining the program and its behavior. As a result of these examinations, the hypothesis is proved true, false, or neither true nor false. Even if you find a hypothesis to be neither true nor false, you have obtained some information about the program or the errors, and you can then proceed with debugging.

♦ Static analysis. To examine program errors, you analyze both syntactic and semantic properties, including program structures and dependencies, cross-references, intermodule interfaces, type consistency, and formal proofs.

♦ Dynamic analysis. In dynamic analysis, you execute the program with appropriate input data and examine its behavior and output. Many people consider correcting errors found through dynamic analysis to be debugging, and often use debugging tools to execute dynamic analysis.

♦ Semidynamic analysis. This technique falls between static and dynamic analysis. It involves hand simulation and symbolic execution and uses a computation model of programs or programming languages.

♦ Program modification. You may modify the program and execute it to verify hypotheses about the relationship between modification and error occurrence. Sometimes a modified program works without error even though you don't know the causes of the error completely.

DEBUGGING VARIATIONS

In our model, the separate processes of localizing bugs and correcting errors are folded into a single loop of hypothesis and verification. This is why the hypotheses in your model could concern anything about the program to be debugged and its errors.

You can derive variations of this model. For example, Figure 2 shows a derived model that separates localization and correction into two successive phases. For example, to debug a nondeterministic concurrent program, you would first locate the bug by finding conditions for reproducing the error and then begin to fix the bug. An algorithmic debugging process in logic programming also has two phases.

DEBUGGING PROCESSES

Our model also relates to traditional debugging processes and support tools.

Debug statements. When you insert debug statements in a program, you have already developed some hypotheses and

are trying to verify them. Based on those hypotheses, you decide where debug statements should be inserted and what information should be printed.

You develop such hypotheses and verify them through abstracting the program (for example, by interpreting the values of variables from the viewpoint of the program specification) or through an examination of an execution of the program following the computation model.

Breakpoints. When you use a debugger and set breakpoints, you also have developed some hypotheses. Basically, such a debugger helps you insert debug statements and examine the program's behavior. As with debug statements, it is not the debugger but the programmers that develop hypotheses and verify them. A symbolic debugger provides the source-code level view for an execution of the program, but it does not provide enough abstraction mechanism to verify the hypotheses nor to suggest the further selection of hypotheses.

Tracers. When you use a tracer, you also have some hypotheses about the program's execution, and you examine the execution to verify your hypotheses. In analyzing the execution, you must have in your mind some execution models, which you use as criteria to examine the program's behaviors. Using such models at various abstract levels, you interpret and analyze the execution to locate the bug or find the causes of the error.

Concurrency. When programmers say that they have almost finished debugging a nondeterministic concurrent program when they realize how to reproduce the error, they imply that they know the true character of the error and its cause when they know how to recreate it. In other words, we know what the bug is. In this case the debugging process is therefore a two-phase process of bug localization and correction.

To recreate the condition that causes an error, you execute the program with a variety of input data and under a variety of runtime environments. To do this, you must first develop some hypotheses, and even if you fail to verify the hypothesis, you can maintain your hypothesis set by adding its refutation to it. By accumulating such refuted hypotheses, you gradually get closer to the error until you realize its cause at last.

You need much experience in system development and a good comprehension of application domains to develop qualified hypotheses from the existing ones and refuted ones, and to verify them by observing the execution of the program.

Algorithmic debugging. An algorithmic debugging method in logic programming is a semiautomated mechanism for lo calizing a bug by using programmers' decisions as oracles.[4] It finds an incorrect procedure based on a standard tracing technique. This process is the localization part of the two-phase process shown in Figure 2.

In algorithmic debugging, a hypothesis that the result of a procedure call is true will be automatically generated and either verified or refuted by the programmers' oracle. According to the result of hypothesis verification by the programmers' decision, a new hypothesis will be generated and examined. Finally, the incorrect procedure will be found.

Today's support tools. Most of today's debugging tools provide mechanisms for observing program execution and helping programmers understand program behavior. Such debugging tools support the verification of hypotheses that programmers develop, but do not support hypothesis generation or selection. Programmers develop and select hypotheses themselves.

High-level debugging facilities support hypothesis verification only at more abstract levels. Source-level debugging, execution visualization, execution replay, enforced execution control, backward tracing, and other techniques are used to verify or refute hypotheses.

Debugging tools must support each stage in the debugging process: hypothesis verification, hypothesis-set modification, and hypothesis selection.

GENERAL FRAMEWORK

We can describe a minimal set of requirements for a general debugging framework in terms of both the theory behind debugging methodologies and the support tools. Since these issues provide the theoretical foundations in a general framework for debugging, and they are needed to establish a debugging methodologies and to develop debugging support environments, the following discussions often mention them and thus will describe them implicitly.

Support tools. Debugging tools must support each stage in the debugging process: hypothesis verification, hypothesis-set modification, and hypothesis selection.

Hypothesis verification. A debugging tool should provide a facility of theorem proving or at least proof-checking to help programmers verify their hypotheses about programs and errors. Hypotheses should be described formally and treated formally in reasoning about location and removal of bugs.

A debugging tool should provide abstract and comprehensive devices to observe and examine program execution behavior. It should provide various viewpoints at various abstraction levels to examine programs and their semantics. It should also be sound — it should not lie about the behavior and the properties of programs.

Hypothesis-set modification. A debugging tool should help programmers maintain the hypothesis set by providing well-defined operations. Because hypotheses can be associated with any program attributes and can be related to other hypotheses and to the properties of errors and verification processes, a debugging tool must serve as a management system for a hypothesis database.

```
procedure CTB is
-- An example of
Complex-Tasking-Blocking deadlock.
  task T1 is
    entry E1;
  end T1;
  task T2 is
    entry E2;
  end T2;
  task LIVELOCK;
  function GET return INTEGER is
  begin
    T2.E2;
    return 0;
  end GET;
  task body T1 is
    task T3;
    task T4 is
      is entry E4;
    end T4;
    task body T3 is
    begin
      B:
      declare
        task T5;
        task body T5 is
        begin
          T4.E4;
        end T5;
      begin
        null;
      end B;
      T1.E1;
    end T3;
    task body T4 is
      task T6;
      task body T6 is
        I:INTEGER := GET;
      begin
        null;
      end T6;
    begin
      accept E4;
    end T4;
  begin
    accept E1;
  end T1;
  task body T2 is
  begin
    select
      when FALSE => accept E2;
    or
      terminate;
    end select;
  end T2;
  task body LIVELOCK is
  begin
    loop
      null;
    end loop;
  end LIVELOCK;
begin
  null;
end CTB;
```

Figure 3. An Ada program with a tasking deadlock.

As a result of hypothesis verification, the hypothesis set will be modified according to the consequence of reasoning from the existing hypotheses and the verification result. This requires nonmonotonic in addition to standard reasoning to maintain the hypothesis set.

Hypothesis selection. A debugging tool should provide an appropriate strategy to select a hypothesis and proceed with the debugging process according to the adopted strategy — or at least give some suggestion for the next selection.

Although the aim of the debugging process is to locate and correct bugs, you sometimes want to verify hypotheses that do not directly concern this ultimate aim. Therefore, you should know the purpose of each hypothesis selection.

Requirements for hypothesis selection include avoidance of infinite loops and accomplishment of the aim of each strategy. A selection strategy should be complete: It should either let you achieve your aim of locating bugs or certify the removal of bugs, or help you find that your strategy is no longer effective so you can abandon it and adopt another.

Theoretical foundations. To establish a systematic debugging method, it is most important to define hypothesis domains in terms of hypothesis characteristics and operations on the domains.

Hypotheses concern anything about programs and errors, so we need to develop theories about a broad range of issues:

- Theories on programs: application domains, formal specifications, computation models, program dynamics, observable equivalences, and program dependencies.
- Theories on errors: classification and causality.
- Theories on reasoning: inference rules, nonmonotonic reasoning, soundness, and completeness.

EXAMPLE TOOL

We have developed an execution monitor, Eden, which serves as a debugging tool within our general framework. Eden is an event-driven execution monitor for concurrent Ada programs.[5] It monitors and records the tasking behavior of a program and provides Ada programmers with facilities to observe, analyze, and understand the tasking behavior. It can also detect tasking deadlocks automatically while monitoring program execution.[6]

Execution histories are indispensable for debugging because hypotheses about bugs and their correction must be developed, selected, verified, and modified based not only on the program and its specification, but also on its erroneous behavior.

Eden follows a program-transformation approach. It consists of a preprocessor and a runtime monitor. The preprocessor transforms a target Ada program, P, into another Ada program, P', such that some entry calls to the runtime monitor are inserted into P to extract information about P's tasking behavior. After the transformation, Eden compiles P', links it with the runtime monitor, and executes it.

We have developed Eden, which is a debugging tool within our general framework. Eden is an event-driven execution monitor for concurrent Ada programs.

During its execution, P' communicates with the runtime monitor when each tasking event of P occurs in P' and passes information about the tasking event to the runtime monitor. The runtime monitor records all collected information into an execution history database and analyzes and reports the tasking behavior of P in response to users' queries.

An Ada programmer can interactively use Eden to observe the tasking behavior of a target program by querying about snapshots of task states and entry queue states, event histories of tasks, and operation histories of entry queues in the program.

The programmer can also use the execution history to do some postanalysis and replay of the program's tasking behavior. When an error is detected (based on the program, its specification, and execution history of erroneous program behavior), the programmer can use information provided by Eden to develop, select, verify, and modify hypotheses about the error until he locates the bug or bugs that caused the error.

Eden detects tasking deadlocks in a target program by examining a directed cycle in a Task-Wait-For graph that is a formal representation of the state of tasking-object interaction in the program.[7]

A Task-Wait-For graph is an arc-labeled, directed graph constructed of two kinds of nodes and five kinds of arcs. Each node represents an executing task, an executing block, or an executing subprogram. Each arc corresponds to each of five types of tasking-waiting relation among tasking objects. By monitoring the tasking behavior of a target program, Eden can create and update a Task-Wait-For graph for the program during monitoring.

For example, the Ada program in Figure 3 results in an tasking deadlock involving all five types of tasking-waiting relations. Figure 4 shows the Eden report when it detects a tasking deadlock. Eden explicitly reports all tasking objects involved in the deadlock, the tasking-waiting relations among the tasking objects, and a snapshot of task states in the program. Using this report, a programmer can locate the deadlock in the program.

A tasking deadlock is an error, not a bug, because which task is the culprit that should be aborted depends on the goal of the program in question. In the example, the location of the deadlock does not mean the location of the bug leading to the deadlock. To locate the bug, the programmer must develop hypotheses about the bug based on Eden's deadlock reports and execution history, and then select, verify, and modify his hypotheses repeatedly by analyzing the program's specification, source code, and execution history until the bug is located.

An execution monitor that extracts, records, analyzes, and reports information

```
***  EDEN Ver. 2.0    9-Jan-1991      date:10-Jan-1991   time:10:35:42  ***

  * A tasking deadlock will occur in the program.

  * At     79.54000000 sec after execution start,
    there is such a situation as follows :

  * task T4 is waiting for activation completion of task T6
  * task T6 is calling entry E2 of task T2
  * task T2 is waiting for terminating together with task T1
  * task T1 is waiting for accepting a call to its entry E1 from task T3
  * task T3 is executing block B
  * block B created task T5 and is waiting for termination of task T5
  * task T5 is calling entry E4 of task T4

***  EDEN Ver. 2.0    9-Jan-1991      date:10-Jan-1991   time:10:36:11  ***

  * At   109.69000000 sec after execution start,
    states of all tasks are as follows :

  * task name              => MAIN_TASK
  * task ID of the task    =>  1
  * state of the task      => BLOCK_COMPLETED
  * calling subprogram     => CTB

  * task name              => LIVELOCK
  * task ID of the task    =>  2
  * state of the task      => WORKING_FOR_INTERNAL_AFFAIRS

  * task name              => T2
  * task ID of the task    =>  3
  * state of the task      => TERMINATION_WAITING

  * task name              => T1
  * task ID of the task    =>  4
  * state of the task      => ACCEPTING
  * communication entry    => T1 . E1

  * task name              => T4
  * task ID of the task    =>  5
  * state of the task      => EXECUTION_WAITING

  * task name              => T3
  * task ID of the task    =>  6
  * state of the task      => BLOCK_COMPLETED
  * executing block        => B

  * task name              => T6
  * task ID of the task    =>  7
  * state of the task      => SIMPLE_ENTRY_CALLING
  * communication entry    => T2 . E2

  * task name              => T5
  * task ID of the task    =>  8
  * state of the task      => SIMPLE_ENTRY_CALLING
  * communication entry    => T4 . E4
```

Figure 4. *The Eden report of the tasking deadlock in the program in Figure 3.*

about the behavior of target programs can support various debugging activities including error reporting, hypothesis generation, hypothesis modification, hypthesis selection, and hypothesis verification. An execution monitor is particularly important for concurrent program debugging because there is no guarantee that the execution behavior of a concurrent program can be reproduced.

Although formal frameworks for testing have been proposed,[8] we need more investigation before we can establish a formal framework for debugging. Unlike testing, debugging concerns not only the program specification and source code, but also — and more essentially — the various causes of errors.

The major subject of debugging is causal reasoning within an iterated process of developing, selecting, verifying, and modifying hypotheses about errors. Therefore, to establish a formal framework we need not only theories on specification, semantics, and behavior, but also theories on error analysis and causal reasoning.

Unfortunately, there has been little work done on error analysis and causal reasoning. We have many subjects to investigate to establish a formal framework for systematic debugging.

We believe debugging is a process of building up hypotheses and verifying them. Knowledge-based facilities show promise to perform debugging under our process model. Such tools may provide smart methods to describe hypotheses and tactics, maintain hypotheses, and employ tactics, among other things.

Our debugging framework is applicable to higher level development stages such as specification and design. Even in these stages, you must validate the specification or design with respect to requirements. If you find any defects in a specification or design, you must "debug" them at the specification or design level. Therefore, when you validate a specification or design, you must consider the issues outlined here.

In general, investigating debugging is really an investigation into the human activities of intelligence and creativity. This makes it both very important and very difficult. ♦

ACKNOWLEDGMENTS

We thank Mikio Aoyama and Itaru Ichikawa at Fujitsu Ltd. and Kazuhiko Yokoyama at Yaskawa Electric Manufacturing for stimulating discussions and valuable comments about debugging methods and tools. We also thank Kazuo Ushijima at Kyushu University for his encouragement and Toshihisa Takagi at Kyushu University for his discussions with us.

REFERENCES

1. "Proc. SIGSoft and SIGPlan Software Eng. Symp. High-Level Debugging," *SIGPlan Notices*, Aug. 1983.
2. "Proc. SIGPlan and SIGOps Workshop Parallel and Distributed Debugging," *SIGPlan Notices*, Jan. 1989.
3. C.E. McDowell and D.P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, Dec. 1989, pp. 593-622.
4. E. Shapiro, *Algorithmic Program Debugging*, MIT Press, Cambridge, Mass., 1982.
5. J. Cheng, K. Araki, and K. Ushijima, "Event-Driven Execution Monitor for Ada Tasking Programs," *Proc. Compsac*, CS Press, Los Alamitos, Calif., 1987, pp. 381-388.
6. J. Cheng and K. Ushijima, "Detecting Tasking Communication Deadlocks in Concurrent Ada Programs," Proc. Int'l Computer Science Conf., CS Press, Los Alamitos, Calif., 1988, pp. 138-145.
7. J. Cheng, "Task Wait-For Graphs and their Application to Handling Tasking Deadlocks," *Proc. Tri-Ada Conf.*, ACM, New York, 1990, pp. 376-390.
8. J.S. Gourlay, "A Mathematical Framework for the Investigation of Testing," *IEEE Trans. Software Eng.*, Nov. 1983, pp. 686-709.

Keijiro Araki is an associate professor of computer science and communications engineering at Kyushu University. His research interests include programming languages, program specification and verification, parallel and distributed systems, and system-development methodologies.

Araki received a B.Eng., M.Eng., and Dr.Eng., all in computer science and communications engineering from Kyushu University. He is a member of the IEEE Computer Society, ACM, IPS Japan, IEICE Japan,

Zengo Furukawa is a lecturer of computer science and communications engineering at Kyushu University. His research interests include testing, debugging, verification, and software-development processes.

Furukawa received a B.Eng. and M.Eng. in computer science and communications engineering from Kyushu University. He is a member of the IEEE Computer Society, ACM, IPS Japan, IEICE Japan, and JSSST.

Jingde Cheng is an associate professor of computer science and communications engineering at Kyushu University. His research interests include specification and programming languages, logical basis of knowledge engineering, and knowledge-based software engineering.

Cheng received a BS in computer science and technology from Tsinghua University, China and an MS and PhD in computer science and communications engineering from Kyushu University. He is a member of the IEEE Computer Society, ACM, IPS Japan, and JSSST.

Address questions about this article to the authors at Computer Science and Communications Engineering Dept., Kyushu University, Hakozaki, Higashi-ku, Fukuoka 812 Japan; Internet araki@csce.kyushu-u.ac.jp.