

Automated Debugging: Are We Close?

Despite increased automation in software engineering, debugging hasn't changed much. A new algorithm promises to relieve programmers of the hit-or-miss approach to isolating a failure's external root cause.



Andreas
Zeller
Saarland
University

For the past 50 years, software engineers have enjoyed tremendous productivity increases as more and more tasks have become automated. Unfortunately, debugging—the process of identifying and correcting a failure's root cause—seems to be the exception, remaining as labor-intensive and painful as it was five decades ago. An engineer or programmer still has to notice something, wonder why it happens, set up hypotheses, and then attempt to confirm or refute them.

In theory, there is no reason to continue this legacy. Debugging can be just as disciplined, systematic, and quantifiable as any other area of software engineering—which means that we should eventually be able to automate at least part of it. So far, research has concentrated on program analysis because of its roots in compiler construction. But analysis requires complete knowledge about the program being examined and does not scale well to large programs.

Testing is another way to gather knowledge about a program because it helps weed out the circumstances that aren't relevant to a particular failure. If testing reveals that only three of 25 user actions are relevant, for example, you can focus your search for the failure's root cause on the program parts associated with these three actions. If you can automate the search process, so much the better.

This is the premise of Delta Debugging, an algorithm that uses the results of automated testing to systematically narrow the set of failure-inducing circumstances.¹ Programmers supply a test function for each bug and hardcode it into any imperative language. The test function checks a set of changes to

determine if the failure is present or if the outcome is unresolved and feeds that information to the Delta Debugging code.

Programmers can either hardcode the algorithm, available from the Delta Debugging Web site (<http://www.st.cs.uni-sb.de/dd/>) or download and adapt the core of Wynot, a prototype debugger written in Python that runs on a Unix or Linux system. Both the algorithm and Wynot (short for “worked yesterday, not today”) fit with any imperative language, and Wynot is platform independent.

Work on a “debugging server” is in progress. The idea is to have a programmer submit the program to be examined along with invocation details to a Web site, choose from a menu how to identify a failure, and click on “Submit.” Wynot will then automatically isolate the failure-inducing circumstances and e-mail the results to the programmer. Plans are for the server to be operational by spring 2002. Eventually, users might be able to run their own debugging server—for example, within a company intranet or as a module in an automated test system.

HOW DELTA DEBUGGING WORKS

As Figure 1 shows, external failures typically come from program input, user interaction, and program changes. Within each of these categories are myriad circumstances, any one of which could be the root cause of the failure.

Delta Debugging requires a test to prove that each circumstance is really failure inducing. An automated testing environment typically provides such a test, but the number of test runs can be quite large. The

trade-off is that, unlike conventional debugging, Delta Debugging always produces a set of relevant failure-inducing circumstances, which offer significant insights into the nature and cause of the failure. Also, if you know something about the structure of the failure-inducing circumstance, fewer tests are required.

How time-consuming it is to incorporate Delta Debugging in a testing environment depends on the circumstances. Program input is typically easy to access, modify, and evaluate. Adapting Delta Debugging to run a program on a different input should take only a matter of hours. Examining user-interaction history requires external tools that record and play back user interaction. Program changes are moderately easy to access and simple to modify, but altering configurations can be difficult, depending on the tools you use.

My colleagues and I used the Wynot prototype to debug Mozilla, Netscape's open-source Web browser project (<http://www.mozilla.org/>), and the GNU debugger. Specifically, we used Wynot to

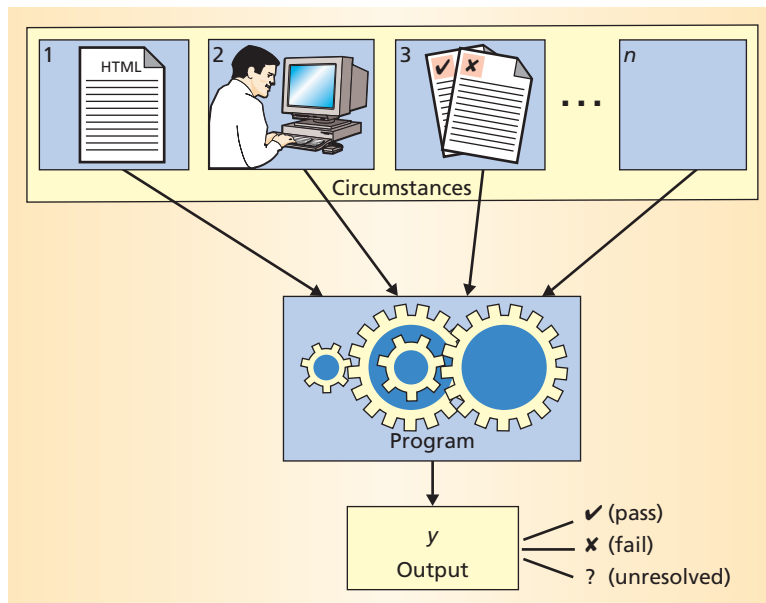
- simplify HTML input that causes the Mozilla browser to fail—after 57 test runs, only one of 896 HTML lines remained;
- simplify Mozilla failure-inducing user interaction—after 82 test runs, we saw that only 3 of 95 user actions were relevant for the failure; and
- identify failure-inducing code changes—after 97 tests, we narrowed a set of 178,000 changed lines to one changed line in the GNU debugger that caused a failure.

SIMPLIFYING PROGRAM INPUT

Mozilla is a work in progress with a wide audience (Netscape 6 uses a Mozilla variant), and Mozilla engineers receive several dozens of bug reports a day. Their first step in processing a bug report is to simplify it—eliminate all details that are irrelevant to the failure. In part, a simplified bug report makes debugging easier because it replaces other reports with irrelevant details.

In July 1999, Bugzilla, the Mozilla bug database, listed more than 370 open, unsimplified bug reports, and the queue was growing. Seeing that Mozilla engineers “faced imminent doom,” Eric Krock, the Netscape product manager, sent out the Mozilla BugAthon, a call for volunteers to help simplify bug reports.² Each volunteer was to turn a bug report into a set of minimal test cases, in which every input would be significant in reproducing the failure. For every five simplifications, a volunteer would get an invitation to the Mozilla launch party; 20 simplifications would earn the volunteer a T-shirt signed by grateful engineers.

The following was Bugzilla entry #24735:



Ok the following operations cause mozilla to crash consistently on my machine

- Start mozilla
- Go to bugzilla.mozilla.org
- Select search for bug
- Print to file setting the bottom and right margins to .50 (I use the file /var/tmp/netscape.ps)
- Once it's done printing do the exact same thing again on the same file (/var/tmp/netscape.ps)
- This causes the browser to crash with a segfault

To simplify bug reports, BugAthon volunteers were supposed to load the Bugzilla Web page (<http://bugzilla.mozilla.org/>) into their text editors and then follow the BugAthon instructions for simplifying Mozilla test cases:

Start removing HTML markup, CSS rules, and lines of JavaScript from the page. Start by removing the parts of the page that seem unrelated to the bug. Every few minutes, check the page to make sure it still reproduces the bug. [...] When you've cut away as much HTML, CSS, and JavaScript as you can, and cutting away any more causes the bug to disappear, you're done.²

This volunteer most likely carried out the process manually, but there is no need to—especially in an environment designed for automation. If you have an

Figure 1. How circumstances affect a program's behavior. A failure can stem from a range of circumstances, including program input, such as an HTML page that makes a browser fail, user interaction that makes a program crash, or changes a programmer makes after the program fails a regression test.

```

1 <SELECT NAME="priority" MULTIPLE SIZE=7> X
2 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
3 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
4 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
5 <SELECT NAME="priority" MULTIPLE SIZE=7> X
6 <SELECT NAME="priority" MULTIPLE SIZE=7> X
7 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
8 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
9 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
10 <SELECT NAME="priority" MULTIPLE SIZE=7> X
11 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
12 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
13 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓

```

```

14 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
15 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
16 <SELECT NAME="priority" MULTIPLE SIZE=7> X
17 <SELECT NAME="priority" MULTIPLE SIZE=7> X
18 <SELECT NAME="priority" MULTIPLE SIZE=7> X
19 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
20 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
21 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
22 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
23 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
24 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
25 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
26 <SELECT NAME="priority" MULTIPLE SIZE=7> X

```

Figure 2. Simplifying a bug in Mozilla's HTML input. The Wynot prototype starts cutting large chunks of HTML input (many characters at a time) and gradually narrows the cut to just a few characters. By line 26, only <SELECT> remains. The rest of the characters (gray) have been cut. The ✓ and X symbols denote test outcome pass and fail, respectively.

automated test that tells whether or not the failure is still present, you can easily automate simplification. Setting up an automated test that checks whether printing a specific HTML page works is not very difficult. All you need is a record-and-replay facility for user interaction.

One approach to automating simplification is to write a program that removes single characters from the input. After each removal, the program would check whether or not the failure still occurs, using the replay tool to automate execution. It would repeat the check until only characters relevant in producing the failure remain.

Unfortunately, for the 40,000 characters of the Bugzilla Web page, this approach would require at least 40,000 tests—and this is the good news. The bad news is that you could end up cutting away the last character again and again.

A better approach is to use the technique experienced programmers use: Cut away large chunks first and increase granularity later. For example, start by cutting away chunks of 20,000 characters, then chunks of 10,000, 5,000, 2,500 characters, and so on, cutting away anything not relevant for the failure. By increasing the granularity, you eventually remove single characters.

This is the approach we took in building our prototype debugger. Wynot starts with large chunks of an input such as replaying a Mozilla user interaction and keeps removing parts until the chunk size is minimal and it can no longer simplify the input.

We reproduced the “Mozilla cannot print” failure as Bugzilla entry #24735 described it and ran Wynot using the Bugzilla HTML code as input. After 57 test runs on a 400-MHz Linux PC, each starting Mozilla and replaying the previously recorded user interaction, Wynot simplified the original 896 lines to a one-line input:

```
<SELECT NAME="priority" MULTIPLE
SIZE=7>
```

Another 25 runs reduced this line to just the <SELECT> tag. Each run took an average 15 seconds,

for a total of about 21 minutes. Figure 2 shows how the algorithm worked.

SIMPLIFYING USER INTERACTIONS

Having simplified the HTML input, we were ready to look at other details in the Mozilla bug report. Was “setting the bottom and right margins to .50” really necessary, for example? In a separate test from the HTML input test, we subjected the log of recorded user interactions to Wynot to find the minimum set of failure-inducing user actions. After 82 test runs (21 minutes), only 3 of the 95 user actions remained:

1. Press the *P* key while holding the *Alt* modifier key. (Invoke the *Print* dialog.)
2. Press *mouse button 1* on the *Print* button without a modifier. (Arm the *Print* button.)
3. Release *mouse button 1*. (Start printing.)

Among the removed actions were moving the mouse pointer, selecting the print-to-file option, altering the default file name, setting the print margins to .50, and releasing the *P* key before clicking on “Print.” All these were irrelevant in producing the failure. Pressing the mouse button before releasing it *was* relevant, however.

Thus, after simplifying both HTML input and user actions, the bug report was

```
Printing a page containing <SELECT> makes Mozilla
crash.
```

That’s it—everything else was irrelevant.

In principle, Mozilla engineers could easily apply this minimization procedure automatically for the 12,479 open bugs (as of 15 February 2001) in the Bugzilla database, as long as they could reproduce bug reports automatically. All that is needed are an HTML input, a sequence of user actions, an observable failure, and a little time to let the computer simplify the failure-inducing input.

Subsequent testing on 44 different input-related failures yielded similar results: Wynot always simplified the input significantly.³ The Delta Debugging Web site offers the public-domain tool we used to

record and play back events for the Unix/Linux X system.

ISOLATING DIFFERENCES

To simplify an input to n characters, Wynot requires at least n tests—simply because it must verify that each character is relevant for the failure. This is a penalty for complex inputs. As an alternative, we tried another approach, which is much more effective: If you know that there is another input where the failure does not show up, you can focus on the input *differences* rather than on the entire input. In general, isolating a small failure-inducing difference is more efficient than simplifying a large failing input.

Figure 3 shows how we isolated a failure-inducing difference in HTML code. Again, for the failing run, we started with a single-line input containing the SELECT tag. The passing run had no input. To narrow the difference between the two, we could have removed differences from the failing run and kept cutting away until a minimal set remained. But rather than simply cutting input while the failure persisted, we also *added* input while the program still passed the test. The combination of cutting and adding rapidly pared down the set of differences whenever a test either passed or failed.

In Test 3 in Figure 3, for example, half the HTML input passes the test. Thus, this input is not failure inducing, and we can include it in all further tests. The difference becomes smaller. In Test 4, Wynot has added half the remaining input. Now, the test fails, which indicates that the failure-inducing difference is in the text just added. The difference becomes smaller still.

Repeating this scheme took only seven tests to narrow the failure-inducing difference to one character. The input of the failing run is reduced to

```
<SELECT NATy" MULTIPLE SIZE=7>
```

while the input of the passing run expanded to

```
SELECT NATy" MULTIPLE SIZE=7>
```

The difference between them is only the first character, the angle bracket, which when missing changes the HTML tag to ordinary text. As in the HTML program input test, the failure depends on whether the user prints a SELECT tag or ordinary HTML text.

FINDING FAILURE-INDUCING CODE CHANGES

The most important source of failure-inducing circumstances is, of course, the code itself. I've shown how Delta Debugging works to isolate failure-inducing program input. In a sense, program code itself is input—input to some machine executing the program. So at least in theory, you could apply Delta Debugging

```
↓ 2 <SELECT NAME="priority" MULTIPLE SIZE=7> X
   4 <SELECT NAME="priority" MULTIPLE SIZE=7> X
↑ 7 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
   6 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
   5 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
   3 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
   1 <SELECT NAME="priority" MULTIPLE SIZE=7> ✓
```

to simplify program code as well—just cut away code until only a relevant slice remains. In practice, however, you would face a major problem: The code of any nontrivial program is far more interwoven than typical program inputs. You can easily cut away some HTML code and still get something meaningful, but removing a piece of code that initializes a variable can dismantle the entire program. Therefore, most changes, such as removing statements, will lead to unresolved test outcomes. This gets you closer to the worst case—a quadratic number of tests.

Additional program analysis, especially program slicing,^{4,5} can eliminate several irrelevant program parts from the start. However, we're only beginning to explore the integration of program analysis and Delta Debugging. In short, simplifying failure-inducing program code is not ready for practice.

Again, a more practical approach is to isolate a difference. If you know that the failure does not show up in another program version, you can have Delta Debugging isolate the failure-inducing difference.

We used this approach to isolate the failure-inducing changes to GDB, the GNU debugger. In July 1998, a Motorola employee sent in a bug report stating that after upgrading GDB from version 4.16 to 4.17, running the debuggee from GNU data display debugger (DDD), a graphical front end for GDB, no longer worked:

Date: Fri, 31 Jul 1998 15:11:05-0500

To: DDD Bug Report Address <bug-ddd@gnu.org>

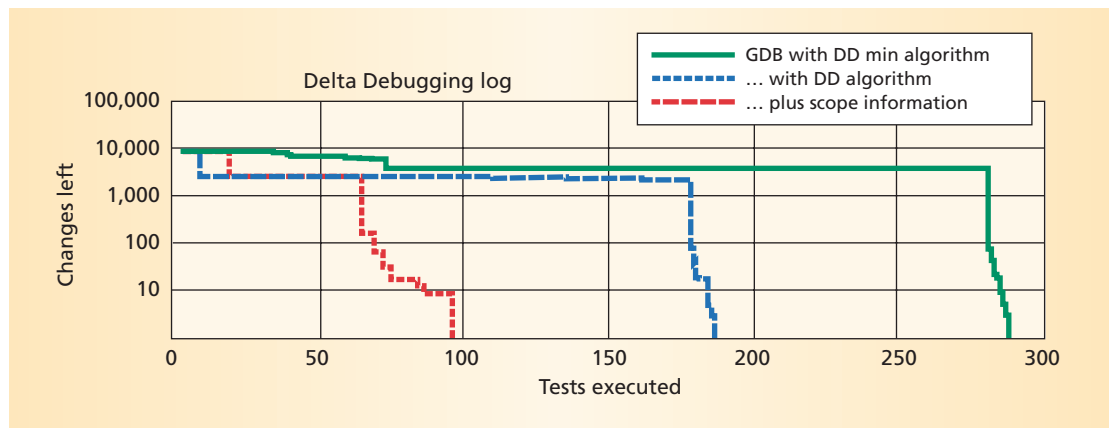
Subject: Problem with DDD and GDB 4.17

When using DDD with GDB 4.16, the run command correctly uses any prior command-line arguments, or the value of "set args". However, when I switched to GDB 4.17, this no longer worked: If I entered a run command in the console window, the prior command-line options would be lost. [...]

This is a classical instance of the "worked yesterday, not today" problem: Some change (or difference) causes a failure. In this case, the change is *huge*: The difference between the source codes of GDB 4.16 and GDB 4.17 is 178,000 lines—178,000 lines have been either added, deleted, or changed between the two releases. (If you consider that the GDB source code itself has about

Figure 3. Focusing on the difference between the passing run (bottom line) and failing run (top line). The combined approach of cutting differences and adding them to passing runs is more efficient than just simplifying. After only seven tests, Wynot isolates the failure-inducing difference—the left angle bracket of the HTML tag.

Figure 4. Delta Debugging (DD) optimizations. Minimizing the set of changes between GDB 4.16 and GDB 4.17 isolates the failure-inducing change after 288 tests. Narrowing the difference between the two versions finds the same change after 187 tests. Grouping changes according to scope reduces the tests to 97.



500,000 lines, and that about a third of the code has been changed, you might wonder why everything else still works.) Somewhere within these 178,000 lines lie the changes that caused DDD to fail—but where?

We used Delta Debugging to isolate this cause. First, we decomposed the 178,000-line diff into 8,721 textual changes in the GDB source, with any two textual changes separated by a context of at least two unchanged lines. We then let Wynot isolate the failure-inducing changes. Each test applied a subset of changes to the GDB code, rebuilt GDB, and ran an automated test that verified whether or not the failure was still present.

Although we continued to use the isolation approach to reduce the differences between the two original versions, we faced a new problem. Wynot had no knowledge about the semantics of the changes and combined them more or less arbitrarily. This caused tests to turn out unresolved because the changes would not integrate, the program would not build, and so on—in short, there was nothing to test. First, we optimized the algorithm to handle unresolved outcomes: By applying smaller and smaller sets of changes, we decreased the risk of conflicting changes. Second, rather than grouping changes arbitrarily, we grouped changes by their *scope*, that is, the directory, file, or function in which they were applied. We assumed that a set of changes was less likely to be in conflict if they all applied to the same scope.

Figure 4 shows the result of these Delta Debugging runs and the effect of the optimizations. Starting with 8,721 changes, Wynot required between 288 and 97 tests—the more optimizations applied, the fewer tests required. On a 400-MHz Linux PC, each test took about 240 seconds to apply the changes and rebuild and run GDB; 97 tests thus required about 6.5 hours.

In all three cases, Wynot reduced the 178,000 lines to the same one-line change that caused DDD to malfunction:

```
diff -r gdb-4.16/gdb/infcmd.c gdb4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged
when it is started. \n\ _____
> "Set argument list to give program being
debugged when it is started. \n\
```

This change in a string constant from arguments to argument list was responsible for GDB 4.17 not inter-operating with DDD. Given the command “show arguments,” GDB 4.16 gives a reply that is obviously constructed from the changed string constant:

```
Arguments to give program being debugged when it
is started is "a b c"
```

GDB 4.17, however, issues a slightly different (and grammatically correct) text

```
Argument list to give program being debugged when
it is started is "a b c"
```

which DDD could not parse. To solve the problem, we simply reversed the GDB change; eventually, we upgraded DDD to make it work with the new GDB version.

We have applied Delta Debugging to find failure-inducing changes in other projects as well. However, in most of these projects, we had a version repository, which lets us group changes by their creation date. With such knowledge, the risk of inconsistencies decreases dramatically. In one case,⁶ Delta Debugging reduced 344 ordered changes from the DDD version repository to a single failure-inducing change—in only 12 tests.

Although Delta Debugging streamlines the process of isolating failure-inducing input and failure-inducing code changes, programmers must still follow the causality chain and decide where to break it. A failure typically does not stem from only one circumstance. Breaking the chain for one failure is easy, but breaking it to eliminate as many failures as possible is challenging.

We are currently applying Delta Debugging to examine internal program states, which isolates even more elements of the causality chain, and first results are promising.⁷ Future combinations of automated testing and program analysis may even assist in properly breaking the causality chain as well.

Also, program input is only one of many circumstances that affect its behavior. The program’s environment, operating system, and runtime library may also be relevant. We are exploring Delta Debugging for circumstances like the following:

- *environment settings* (which part of the environment is relevant?),
- *system libraries* (after an upgrade, I am stuck in a DLL misconfiguration—why?), and
- *thread schedules* in parallel programs (which part of the schedule causes a nondeterministic behavior?).

As we discover more about the structure of these circumstances and the resulting causality chain, we come closer to passing much of the boredom and monotony of debugging onto machines. Eventually, debugging may become as automated as testing—not only detecting failures, but also revealing how they came to be. ★

Acknowledgments


Steven Barlow, Holger Cleve, Kerstin Reese, Torsten Robschink, Gregor Snelting, Falk Schreiber, and the *Computer* reviewers provided helpful comments on earlier versions of this article.

This work was conducted at the University of Passau, Germany, and funded by grant Sn 11/8-1 from Deutsche Forschungsgemeinschaft.

References

1. A. Zeller, “Simplifying and Isolating Failure-Inducing Input,” to appear in *IEEE Trans. Software Eng.*, <http://www.st.cs.uni-sb.de/dd/> (current Oct. 2001).
2. “The Gecko BugAthon,” Mozilla, <http://www.mozilla.org/newlayout/bugathon.html> (current Oct. 2001).
3. R. Hildebrandt and A. Zeller, “Simplifying Failure-Inducing Input,” *Proc. ACM SIGSOFT Int’l Symp. Software Testing and Analysis (ISSTA)*, ACM Press, New York, 2000, pp. 135-145.
4. F. Tip, “A Survey of Program Slicing Techniques,” *J. Programming Languages*, Sept. 1995, pp. 121-189.
5. M. Weiser, “Program Slicing,” *IEEE Trans. Software Eng.*, July 1984, pp. 352-357.
6. A. Zeller, “Yesterday, My Program Worked. Today, It Does Not. Why?” *Proc. Joint 7th European Software Eng. Conf. ACM SIGSOFT Symp. Foundations of Software Eng.*, O. Nierstrasz and M. Lemoine, eds., vol. 1687, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1999, pp. 253-267.
7. A. Zeller, “Isolating Cause-Effect Chains from Computer Programs,” <http://www.st.cs.uni-sb.de/dd/> (current Oct. 2001).

Andreas Zeller is a software engineering professor at Saarland University in Germany. His research interests include dynamic program analysis, configuration management, and program visualization. Zeller received a PhD in computer science from the Technical University of Braunschweig, Germany. He is a member of the IEEE Computer Society and the ACM. Contact him at zeller@computer.org.



cluster computing
 collaborative computing
 dependable systems
 distributed agents
 distributed databases
 distributed multimedia
 grid computing
 middleware
 mobile & wireless systems
 operating systems
 real-time systems
 security

IEEE

Distributed Systems Online

computer.org/dsonline