The background is a vibrant, abstract composition of yellow and green streaks and lines, suggesting motion and energy. Overlaid on this are faint, technical diagrams and circuit-like patterns, including grids, rectangles, and circular elements, which contribute to a high-tech or scientific aesthetic.

Hard-to-track bugs can emerge when you  
can't guarantee sequential execution. The right tools  
and the right techniques can help.

# Debugging in an Asynchronous World

MICHAEL DONAT, SILICON CHALK

**P**agers, cellular phones, smart appliances, and Web services—these products and services are almost omnipresent in our world, and are stimulating the creation of a new breed of software: applications that must deal with inputs from a variety of sources, provide real-time responses, deliver strong security—and do all this while providing a positive user experience. In response, a new style of application programming is taking hold, one that is based on multiple threads of control and the asynchronous exchange of data, and results in fundamentally more complex applications.

But we've all dealt with complex and challenging software before—optimizing compilers, spreadsheets, word processing, text rendering, airline reservations, and space-

probe image enhancement. What's different about our modern asynchronous world that makes this particular type of software so difficult to develop? At Silicon Chalk, we refer to that difference as *emergent behavior*—that is, behavior arising from interactions between components (qualities not observed when working with a component in isolation).

Traditionally, software behavior results from the sequential execution of instructions. Transitions in behavior are easy to identify and understand because we have a sequence that we can reference, like a map. In the asynchronous world, however, software behavior is less tangible, and is the result of several sequences of instructions operating in parallel and communicating with each other. The transitions of individual pieces remain as clear

# Debugging in an Asynchronous World

as before, but transitions of the system as a whole become far more complex as the system-state space is the product of the states of its components. What's worse, we don't have a map to help us.

Emergent behavior creates a host of challenges. How do you go about understanding asynchronous code? Although experienced programmers are adept at executing code in their heads, only a rare few have the ability to mentally execute multiple streams of code and track their interactions. Indeed, we've yet to meet such a person—and would hire them instantly if we did!

From a testing perspective, emergent behavior creates some difficult-to-reach hiding places for bugs. The major challenge is devising strategies to flush bugs out of such problem areas as race conditions, crippling invocations during periods of vulnerability, or confusion over interleaved communication with multiple clients, to name just a few.

And as if a whole new source of difficult-to-reason-about bugs was not bad enough, it is also tremendously difficult to deterministically re-create deviant behavior with instrumentation sufficient to accurately diagnose and repair the fault. Of course, we've faced increasing complexity before. We've progressed from looking at core dumps, to primitive debuggers, to the current crop of symbolic debuggers, memory usage tools (e.g., Purify and BoundsChecker), profiling tools (e.g., gprof and Quantify), and the like, which provide increasingly sophisticated mechanisms for observing, interrupting, and even modifying code as it executes. As in the past, our ability to create larger and more complex code has outstripped the tools at hand for managing and understanding that code. Few tools have the capabilities to deal with large numbers of threads and complex control flows spanning threads, processes, and machines, not to mention the emergent behavior that results.

In our own debugging effort, we rely on liberal use of assertions [1] and the inspection of log files generated

from test runs of the application. Because we expect to see bugs rarely, and know they're hard to reproduce, we need the capability of trapping and examining bugs when they first occur.

From a maintenance perspective, emergent behavior implies the inability to successfully choose between implementation alternatives until you've experienced the behavior of the system as a whole.

After a number of years developing and delivering distributed and realtime applications, we've learned the hard way that such issues cannot be ignored. We'll discuss some of our experiences and provide a new way of looking at the process of debugging, testing, and maintaining asynchronous applications. We'll also look at the shortcomings of existing tools, present some of the mechanisms that we've found useful, and observe ways that tools and approaches may evolve to help meet these challenges.

## IN THE BEGINNING

Before focusing on testing and debugging asynchronous applications, we must mention a few design principles. And while it has always been true that spending time up front creating a good design results in software that's easier to test, debug, and maintain, it is even more important in our asynchronous world.

Most people find it challenging to model emergent behavior. It is important to create points of reference by making those aspects of the design more explicit.

One useful technique is the creation of sequence diagrams for various canonical tasks or activities in the system. Sequence diagrams explicitly represent multiple execution streams, so they provide an ideal medium for asking questions like, "What happens if event B occurs before event A?"—even though the diagram may show them occurring in the opposite order. It's very easy to design a system based on one's narrow vision of how things *should* happen. Sequence diagrams make it easier to start understanding what can happen. We have seen a number of situations in which relatively serious potential flaws in a system have been detected and dealt with during design time through this approach.

Another very useful technique is to identify and explicitly code state machines [2] where possible. No matter how they are written, components move from state to state depending on the communications they receive. When code evolves unguided, these state decisions often are distributed throughout the code as collections of apparently unrelated if statements. Without a *guiding* structure, the state machine that corresponds to the code



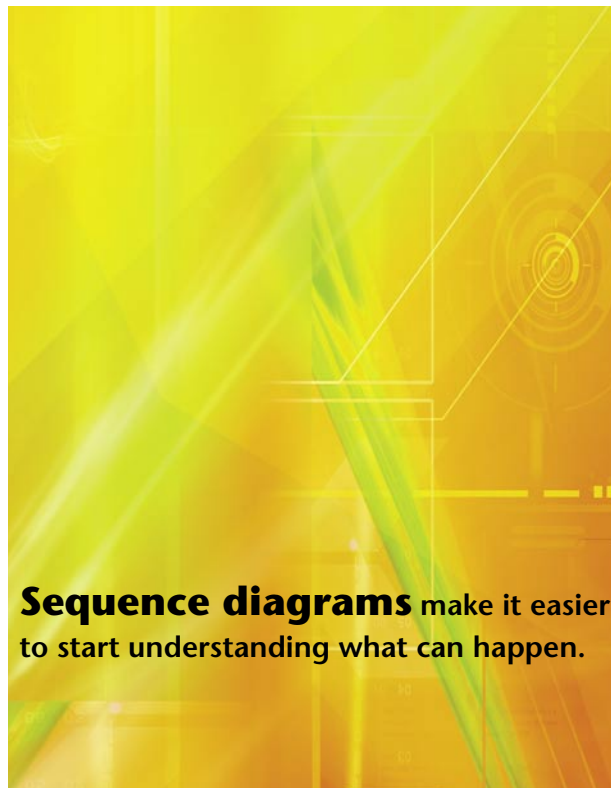
becomes overly complex and error prone, and eventually the code will not be maintainable.

Viewing components as state machines from the beginning structures their evolution along a path that is easier to understand. It provides a central location that encapsulates behavioral decisions. And, like sequence diagrams, state machines are ideal for considering what-if scenarios such as “What if event A arrives when the machine is in state  $S_1$ ?”

A third technique we have successfully applied as part of software development is the aggressive use of assertions. Assertions are useful in a couple of ways. First, they allow programmers to document their assumptions about the circumstances under which their code will be executed. Because they are part of the programming environment, assertions are both concise and precise, and there is a strong motivation to make this form of documentation consistent with the code itself—that is, the code won’t run unless you do.

Second, assertions capture erroneous behavior much earlier than would be the case without them. In their absence, code merrily goes on being executed, even though fundamental assumptions inherent in the code are no longer true. These could range from the inherent assumption that a pointer is non-null when you de-reference it to more complex assumptions about the relationships of various state values. Regardless, by the time the failure is actually noticed (perhaps because you have received an access violation), it can often be far removed from when the assumption was first violated. Because of complex component interactions, it may be next to impossible to reconstruct when and how the assumptions were violated.

The main point of raising these examples is to draw attention to the importance of solving problems early. If you’re in the process of creating an asynchronous application, look around for design techniques that will help



highlight emergent behavior. If much of the application can be expressed as state machines, it could be useful to simulate them separately from the actual coding of the application. Remember that the worst bugs will be difficult to reproduce deterministically; there will be a lot more of these than you are accustomed to in asynchronous applications. In our experience, mechanisms that one puts in place beforehand will help analyze what went wrong afterward (the modern equivalent of reading a core dump), and will help isolate these types of bugs more quickly.

## TESTING

One component of an application that we’re currently working on is a “Directory Service,” which synchronizes data with its counterparts on other machines. The synchronization algorithm is designed to be fault-tolerant over a potentially poor wireless network, while also attempting to be efficient. We’ll use this example throughout the following discussion.

After creating a wonderful piece of asynchronous software, such as our Directory Service, and hopefully catching many of the problems up front as a result of good design, you’ll still need to test your software and make sure it really does what the requirements say it should. There are many different kinds of tests—unit tests, system tests, stress tests, performance tests, and so forth. This discussion will examine the technologies for performing tests rather than the merits of different kinds of testing. We’ll simply focus on:

- How the test stimulus is generated.
- How the application response is evaluated.

When we test our Directory Service, we perform the usual manual testing—human-driven test stimulus and visual evaluation—but we also use scripts to generate test stimuli and evaluate the response by examining log files.

Because applications have become feature-rich, manual testing is simply too irregular, slow, and expensive. We need an automated testing infrastructure.

# Debugging in an Asynchronous World

Most operating systems support test-scripting tools that usually combine a scripting language with some UI-based stimuli generation primitives, together with UI-based response examination primitives. Also available are management capabilities for running scripts, and recording and reporting results—many of which are quite full-featured and can even run applications on a group of machines. Information about these types of tools can be found online, for example, at ApNet (<http://www.aptest.com/resources.html#info-misctools>). Note: You may have a more difficult job finding the appropriate off-the-shelf tools for applications that are not UI-based (such as signal processing).

Components can be tested in isolation using unit-test facilities such as JUnit, or with ad hoc collections of code that use each component in isolation. The test stimulus is simply a direct call to the code under test, and the response is examined via the return value or by probing a data structure. These tests are very useful prior to integration, but are not effective in uncovering the kinds of bugs hiding within emergent behavior; by definition it's a post-integration entity.

The problem with these techniques is that we're only looking at the tip of the iceberg. We're hoping that any bugs will come into view. To adequately address emergent behavior, we need methods that delve deeper as they examine the multi-machine state space. Current test tools aren't

equipped to evaluate asynchronous application state transitions.


In the asynchronous world, the water is a lot deeper. You find the same fish as before near the surface, but now there are many others in the depths that are difficult to spot (and some have really big teeth!). We need to figure out how to net the big ones. We need to examine the depths of the complex state space of asynchronous applications.

During testing, the big questions are: Did everything happen the way it was supposed to happen? Is it enough to look at results via the UI? In our Directory Service example, instances of the application may communicate successfully with each other during a particular test, but how do we know that the synchronization algorithm is operating correctly? Evaluating system response at the UI level isn't good enough; it's possible that the scenario

just performed worked out, but not as expected. In other words, the scenario has uncovered a bug, but you won't see it if you're just looking at the UI. For example, a Directory Service that believes it is synchronized after hearing the same data three times is clearly not working correctly, but the application still appears to function.

When dealing with components that process message streams, it's easy to forget to ask why you're receiving these particular messages. When you look at how instances interact with each other, you may see that messages in item/value pairs have been produced and seem "syntactically" correct, but they don't make sense in the context of the "conversation" between the directories. In order to observe this, you must be able to "hear" the whole conversation, which means examining information from all of the instances.

To determine whether the application is behaving the way we expect, we need to do more than look at the system outputs. In fact, we must do more than probe some of the application's data structures. We have to examine a trace of the sequence of events. We find this to be the norm rather than the exception. In general,



**An interesting part of creating system frameworks is figuring out which interactions to capture—and how to automate the process.**

we should examine interactions between components, for example, identifying at a *global* level whether event A is followed by event B, which is followed by event C—even though each might have been produced by a different component on a different machine. Unfortunately, there is no single component that we can look at to make this assessment.

**Detecting Deep Bugs.** In order to detect bugs during testing, we have found it beneficial to examine the application with tools traditionally used for debugging. The challenge in using these tools is that most debugging tools are oriented toward inspecting and following the activity of a single thread of control; we need to examine interactions between threads of control.

The mechanism that we have found most useful for this is tracing. Of course, tracing facilities are nothing new. Indeed, the old standby of `printf` debugging is just tracing achieved through the manual insertion of trace code. The asynchronous world raises the bar on tracing in several ways:

1. We need to be able to explicitly correlate activity across multiple threads of control, which may even reside on separate machines.
2. Because of the complexity of asynchronous applications, we can't rely on manual intervention for the insertion of tracing code; it's just too labor intensive.
3. As systems often deal with information in realtime, the tracing facility itself must have minimal impact on performance. This is critical, because if execution time is altered, the tracing facility may hide errors sensitive to such things as race conditions.

We believe there are three general approaches to each of these issues. First, the ability to correlate activity across threads of control can often be handled through what we refer to as “interpositioning techniques.” Roughly speaking, these techniques capture interactions between components by interposing themselves in the communications paths between them. There are many different styles of interpositioning, depending on the manner in which components communicate with one another. We'll talk about some of the techniques that we know about, but you'll still want to think carefully about this if you're going to build this sort of tracing facility.

Second, manual insertion of code is addressed in the obvious way: We have to create mechanisms that allow tracing code to be automatically inserted, and ideally, can enable and disable tracing based on factors such as what components are communicating or the particular type of interaction being traced.

Finally, we have to take care in the design of the tracing system itself to minimize alterations in program behavior.

**About Tracing.** Let's start our look into tracing techniques with this last issue. Here are some ideas on minimizing the impact of tracing:

1. Keep the trace mechanism as simple as possible so you can minimize the number of OS calls you have to make.
2. Collect the trace in memory to make it as fast as possible.
3. Use a separate low-priority thread to write trace memory to disk.
4. Temporarily increase the process priority when producing trace output to avoid using locking, which requires a more expensive OS call.
5. Find ways of compressing the trace output in order to minimize disk accesses. One way is to write tokens for subsequent uses of format strings, and record the format string and parameters separately, rather than taking the time to format the trace output during collection.

Tracing everything will likely corrupt our results (through altering the timing of the execution of the code), and we'll want to focus on different aspects of the application for different purposes. Tracing capabilities are also important for debugging activities—diagnosing the cause of bugs found in testing. All too often a symptom appears in one place, but its cause is in another. We also need a mechanism of reconfiguring trace collection with minimal effort in order for the tool to be effective. Large code bases and many independent code paths make manual trace reconfiguration unrealistic. An automated solution is necessary.

As noted earlier, we have had very good experiences using interpositioning techniques to capture relevant trace information. With interpositioning, you have to start by looking at how components interact in your system, and then try to determine the best mechanism for intercepting and tracing those interactions. There are myriad ways components can interact: function calls, method invocations, shared memory, locks, semaphores, event queues, and network messages, just to name a few. We'll start by looking at a concrete example drawn from our own experience and then briefly touch on some others of which we are aware—and that may have widespread applicability. However, systems displaying emergent behavior tend to have idiosyncratic communications mechanisms, and one of the most interesting aspects of creating a framework for your system will be figuring out

# Debugging in an Asynchronous World

which interactions to capture and how to automate the capture process.

One of the applications that we work on is based on COM (Microsoft's Component Object Model system). Like most component systems, COM is based on declarative interface specifications, and provides an infrastructure for the creation of components and lifetime management capabilities.

Fortunately, COM provides the basics necessary to interpose oneself in the communication path. In our case, we take the interface specifications for all components in the system and mechanically post-process them to create new implementations for those interfaces. The purpose of the new implementations—which we refer to as “tracing proxies”—is to trace out information about when invocations occur, who made them, who is receiving them, and the values of the parameters, and then to pass the invocation along, *untouched*, to the real implementation.

At runtime, we can optionally insert a tracing proxy in front of any component in the system. Modifying the component creation facility in COM does this, so it is completely transparent to the application code itself. All invocations directed to a particular component are, in fact, sent to its proxy. The proxy traces out the necessary information, and then makes the corresponding invocation on the real component.

How easy it is to apply interpositioning to other systems? Obviously it depends entirely on the nature of the communication mechanism used and the facilities your OS provides, but there are certainly many situations that we are aware of where this type of approach is viable:

**1. Network.** Tools such as `tcpdump` (Unix) and `netmon/netcap` (Windows) already allow one to interpose oneself in the communications of components that are connected via a network. However, if you have built your own application-level protocols, you might want to consider extending the capabilities of these tools, or even writing your own tools that trace out information containing rich

semantic detail relevant to the interactions. For example, `tcpdump` will give you a dump of the bytes in a User Datagram Protocol (UDP) packet, but you can post-process a `tcpdump` log to interpret those bytes the way your code would and generate a much higher-level trace.

**2. Object dispatch systems.** Both .NET and Java provide facilities at the virtual-machine level to intercept all method invocations.

**3. Component systems.** COM, CORBA, and EJB are based on interface specifications that allow the construction of proxies, as described earlier.

**4. Aspect-oriented programming.** This provides a rich vocabulary for identifying interpositioning points and inserting code.

Traces contain a lot of valuable information. Separating the wheat from the chaff, however, can be overwhelming if just done manually. Searching and filtering with tools such as `grep` and `sed` are quite helpful, but recognizing patterns of communication requires a slightly more sophisticated tool.

We use the term “path expressions” to express the types of behavior we are interested in flagging. We may wish to detect a faulty sequence of exchanges, or we may want to find a particular sequence so we can examine it further. Path expressions are regular expressions that describe these sequences. They are slightly different from those found in tools such as `grep` since we want to ignore any chaff communication packets that might happen to pepper our trace. (In other words, each character *X* in a path expression is implicitly separated by a preceding `grep`-ism: `[^X]*`.)

The path expressions are used to evaluate the trace logs. If the trace doesn't match the path expression, we've detected a bug. But not only that, we also have the context of that bug, which gives us quite a head start in understanding what might be wrong, or how we can reproduce the problem for further study.

Our application has a recording capability that lets users replay the previous interactions with the software. This feature is implemented by storing some of the actual internal events of the application. By leveraging this technology, we have incorporated a scripting mechanism into the application.

Scripts are written in a very simple language. We don't have the loops or if-then-else structures that many scripting tools provide. The only capability we have is to include files, macros, UI primitives, and string concatenation on macro parameters. However, this has provided us with the means of scripting an incredibly wide range of interactions.



Scripts are compiled into recording files containing special events that we call “Monkey” events. The recordings can be played back via a command-line argument, and the Monkey events are converted into their intended UI primitives.

Remember that Directory Service we mentioned earlier? It allows us to have one machine feed script commands to others so that one script can contain a synchronized sequence of UI actions to be performed by a set of machines. This really helps us in such tasks as automatically verifying our latest build and reproducing difficult interactions.

**Debugging.** Programmers are notoriously bad at simulating processors. The only way to know what some non-trivial code will do is to execute it and watch what happens. Often one will see sequences of events that are unexpected and obviously wrong. The same tracing facilities used in testing can aid debugging. However, additional considerations arise in the context of asynchronous debugging.

The most important one is reproducibility. The most difficult bugs to resolve are those that are non-deterministic and hard to reproduce. Attempting to reproduce these bugs by manually working the UI is challenging. Scripted manipulation of the UI is more effective since one can precisely script sequences of UI actions, and vary timing parameters until the bug is observed consistently. You can also vary the order in which events occur to see if the bug is related to interleaved communication among different components. Once you’ve got it reproduced, you’ve also got a test asset moving forward. That script can be reused in the future to make sure the bug will be detected if it is reintroduced.

The assumption in scripting is that the scripting tool can consistently execute script commands within a suitable interval. Synchronizing events on separate machines suffers from interval issues depending on several factors, including network speed and processor load on the machines involved. As a result, there will always be some small interval beyond the capabilities of scripting

software that will cause some bugs to be intermittently reproducible with this technique.

Formal analysis of the communicating state-machine approximations of suspect code might help in understanding some bugs—but it’s a very expensive undertaking, and there is always a gap [3] between the formal model and reality that leaves room for doubt. Formal proofs of code correctness are equally untenable, at least for commercial-grade software.

Since there will always be bugs that are hard to reproduce, our approach is to perform all testing with instrumented versions of our application. This allows us to examine log files once a bug is detected. Even the release version of our software produces log files (log files record only the events of the past hour or so, to ensure that they use a bounded amount of disk space). We justify this as a reasonable strategy because the code faults usually have

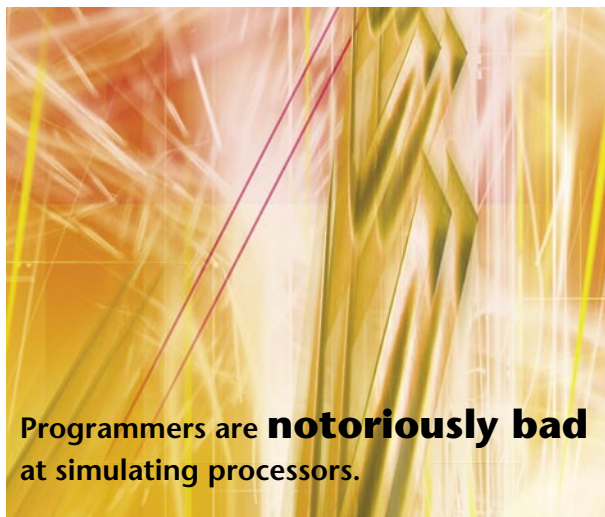
more than one means of displaying themselves. It is likely that the effort required to find the bug in an instrumented version versus a non-instrumented version is almost identical.

Debuggers are essential. Unfortunately there isn’t much support for tracing the multi-machine state space. Simply tracing a logical thread of execution can be very difficult. Once you’ve reproduced the bug, you’re tracing through the code, hot on the trail. You then come to a remote

procedure call, or reasonable facsimile, and the debugger can’t help you trace into the code on the remote machine—dang!

There are two sides to this coin. On the one hand, one would like to be able to trace the logical thread of execution transparently, rather than deal with additional debuggers on remote machines—along with the intricacies of RPC (Remote Procedure Calls) support in dealing with the physical execution contexts and so forth. On the other hand, one would also like to be able to set breakpoints on the service entry point and to examine the stack frame back into the calling client.

As an example in our case, and forgive us for the gory details here, when you make an interface call from a single threaded apartment (STA) component to a multiple



**Programmers are notoriously bad at simulating processors.**



# Debugging in an Asynchronous World

threaded apartment (MTA) component, you can start following the execution in the STA component, but then it disappears into the COM infrastructure and you have no easy way to connect it with the resulting sequence of execution in the MTA component.

Visual Studio .NET supports both logical thread tracing and remote stack frame examination for Web services. For other platforms/environments that don't support these capabilities, one might consider a partial solution. One such solution is to explicitly pass a machine + thread identifier as the first argument to each remote procedure call. This must be part of the design from the start as it is difficult to implement post hoc.

Today's debuggers have fairly good conditional breakpoint capabilities. In an asynchronous system, however, the "event" one is looking for might be a series of code executions in independent execution contexts. Conditionals on breakpoints are usually limited to references on the local machine, so setting the appropriate breakpoint can be a challenge. This is another situation where post-processing traces is more attractive. A trace can be searched and then examined in the vicinity of occurrences of the "event."

An interesting thing about debugging asynchronous applications is that you rely less on the debugger. As these applications become more common, development tools will undoubtedly catch up. In the meantime, you should find ways to build debugging capabilities right into the applications.

We described one technique that has served us very well: flexible tracing based on automatically interposed code. Others that we also considered include deadlock detection by instrumenting locking code, and automatic monitoring of queue lengths (many of our components queue events, and a large backlog of events generally indicates that something has gone wrong). Whenever you discover a class of bugs in your system that is difficult to track down, there's a potentially large upside if you can

figure out how to make your code detect that type of bug, or even point a finger in its direction.

## SQUASHING EMERGENT BUGS

New types of devices, along with the pervasiveness of the Internet, are creating demands for software able to handle a variety of inputs and outputs in realtime. Often, in response to this, applications are structured as a collection of components able to communicate asynchronously. While this architecture helps address the requirements, it also creates systems that exhibit *emergent* behavior that can't be observed when working with a component in isolation.

We discussed some pitfalls of emergent behavior, particularly as it affects testing and debugging. We also touched on a number of problems that we face in our own development efforts, and described how building debugging support directly into the application can help address those problems. Undoubtedly, your experience will be different, but hopefully we provided helpful insights on the problems you might face—as well as a means of dealing with them. ☺

## ACKNOWLEDGMENTS

I'd like to thank my dear colleagues at Silicon Chalk and the ACM reviewers who helped me with this article.

## NOTES

1. Assertions are checks embedded in the debug version of the application that bring up a message box at the first sign of trouble.
2. A state machine describes behavior by explicitly identifying states and the possible transitions between states. Visible behavior is the result of sequences of transitions.
3. The gap exists because the formal model is always someone's interpretation of the code, or vice versa, depending on the technique used. With effort, the gap can be reduced, but it is always present.

## LOVE IT, HATE IT? LET US KNOW:

[queue-ed@acm.org](mailto:queue-ed@acm.org) or [www.acmqueue.org/forums](http://www.acmqueue.org/forums)

**MICHAEL DONAT** is currently director of quality assurance at Silicon Chalk, Vancouver, BC, Canada. He has a Ph.D. in computer science from the University of British Columbia and has been interested in software development issues since working for Microsoft from 1987 to 1992.

© 2003 ACM 1542-7730/03/0900 \$5.00.