PARASOFT®

We make software work.™

When, Why, and How: Code Analysis

With Adam Kolawa, Ph.D.

Parasoft delivers quality as a continuous process

This is the first in a series of interviews in which Adam Kolawa—Parasoft CEO and *Automated Defect Prevention: Best Practices in Software Management* (Wiley-IEEE, 2007) co-author—discusses why, when and how to apply essential software verification methods. The series will also address code review, unit testing, memory error detection, message/protocol testing, functional testing, and load testing.

What do you mean by "code analysis"?

I mean statically analyzing code to monitor whether it meets uniform expectations around security, reliability, performance, and maintainability. Done properly, this provides a foundation for producing solid code by exposing structural errors and preventing entire classes of errors. At Parasoft, we've found that the most effective code analysis encompasses pattern-based (rules-based) static analysis, data flow static analysis, and code metrics calculation.

Let's take a closer look at those three breeds of static analysis. First off, pattern-based static analysis. What is it and why is it valuable?

By pattern-based static analysis, I mean scanning the code and checking whether it has patterns known to cause defects or impede reuse and agility. This involves monitoring compliance to coding standard rules—rules for preventing improper language usage, satisfying industry standards (MISRA, JSF, Ellemtel, etc.), and enforcing internal coding guidelines.

If you nip these issues in the bud by finding and fixing dangerous code as it is introduced, you significantly reduce the amount of testing and debugging required later on—when the difficulty and cost of dealing with each defect increases by over an order of magnitude.

Many categories of defects can be prevented in this manner, including defects related to memory leaks, resource leaks, and security vulnerabilities. In fact, simply using static analysis to enforce proper input validation can prevent approximately 70% of the security problems cited by OWASP, the industry-leading security community.

What's data flow static analysis and why is it valuable?

Data flow static analysis statically simulates application execution paths, which may cross multiple units, components, and files. It's like testing without actually executing the code. It can automatically detect potential runtime errors such as resource leaks, NullPointerExceptions, SQL injections, and other security vulnerabilities. This enables early and effortless detection of critical runtime errors that might otherwise take weeks to find.

While pattern-based static analysis is an error prevention practice, data flow static analysis is an error-detection practice. Like all error-detection practices, it's not 100% accurate and you can't expect that it will uncover each and every bug lurking in your application.

The main difference between pattern-based static analysis and data flow static analysis is that with pattern-based static analysis, you can absolutely guarantee that certain classes of defects will not occur as long as you find and fix the coding constructs known to cause these defects. With data flow static analysis, you are identifying defects that could actually occur when real application paths are exercised—not just dangerous coding constructs. But you have to realize that you will inevitably overlook some bugs, and might have a higher ratio of false positives than you encounter with static analysis.

If data flow static analysis can't find all the bugs, how do you automatically detect the remaining bugs?

Unfortunately, you can't. Parasoft has spent 20 years investigating how and why errors are introduced into software applications. We've found that only certain types of errors can be detected automatically. Most bugs are related to poorly-implemented requirements, missing requirements, or confused users, and cannot be identified without involving human intelligence. There is no silver bullet. With continuous regression testing using a robust set of technologies, you can automatically determine when a modification impacts application behavior, then leverage human intelligence to determine if the change was intentional or not. However, that's the topic of another paper.

The fascinating thing about data flow static analysis is that it's the only automated technology I know of that can actually help you find missing requirements. For instance, assume that data flow static analysis identified a NullPointerException in a Java application. If the architect examines the path that led to the exception, then considers the conditions under which the program might end up going down this path, he might find a missing requirement. Perhaps the exception could be caused by a certain situation that's feasible, but code to handle this exception was never implemented because nobody anticipated that the exceptional situation would occur. Or, maybe someone previously implemented a branch of the code that handled this condition, but it was mistakenly stubbed out or removed during refactoring. This is just one example of how data flow static analysis helps you make some really interesting discoveries about your code.

What about metrics? Why should development teams worry about metrics?

Well, metrics have an interesting history in static analysis. For the most part, they have been ignored. And in my opinion, they should be… except under certain circumstances.

Assume I have a million lines of code. If I find out that the code has an average Cyclomatic Complexity of 9 or 6, what does this tell me? Absolutely nothing.

Since overly-complex code has time and time again been proven to be more error-prone than simpler code, I would appreciate an easy way to zero in on complex code—say, any class or method whose Cyclomatic Complexity is 10 or higher. Parasoft provides traditional metrics calculations just in case someone wants to review them. However, we think that there's more value in our ability to flag code that exceeds industry-standard or customized metrics thresholds.

With an easy way to hone in on what you consider to be overly-complex code, you can ensure that it's examined during peer code review.

Interesting… metrics as an input to peer code review. Do the other types static analysis also relate to code review?

The other types of static analysis are also valuable inputs for peer code review. If a developer does not agree that a certain rule violation should be fixed (e.g., because they don't think that the rule applies in the given context), then the team can use the code review to discuss whether this violation is a prime candidate for a "suppression." Also, code review can be a good opportunity to analyze the defects reported by data flow static analysis; in particular, to determine if they might be indicators of missing requirements, as I mentioned earlier.

Another important point to note while we're talking about the relationship between static analysis and peer code review: when teams are truly committed to running static analysis with a rule set customized to suit their policies and priorities, it eliminates the need for line-by-line inspections during peer code reviews. Reviews can then focus on examining algorithms, reviewing design, and searching for subtle errors that automatic tools cannot detect. This really takes a lot of the drudgery out of the peer code review process—making it not only more productive, but also more engaging for developers.

I suppose that means static analysis should be performed prior to peer code review. Any other tips as to when it should be performed?

Yes—it's essential that it's done continuously, but phased in incrementally.

Some people think they can take the "big bang approach": scan the entire code base once, and then clean up all the reported problems before some milestone, like a release. I've seen many people try this, and it always fails. Why? It's just too overwhelming.

Realistically, if you want to ensure that code conforms to your organization's designated coding policies, you need to make static analysis a continuous yet unobtrusive process. To start, you configure your static analysis tool so that every day, it automatically scans the project's code base (from source control), checks adherence to your policy, and reports the results to the developers' IDEs. The trick is that instead of having it check the entire code base, you just have it check code that was added or modified since the "cut-off date," which is the date when you introduced this initiative. Otherwise, the developers are overwhelmed by the sheer number of reported violations, and don't end up fixing any of them. If you focus on the code that people are really working on and require that violations are fixed within one or two days of introduction, then the code base will slowly but surely start conforming to your policy.

This incremental approach goes against what almost everyone in the industry tries to do, especially with security. As the code is being developed, it needs to be brought under the policy—security policy, quality policy, whatever—every single day. If they keep on top of it, it hardly impacts the team's workflow at all. In fact, it actually saves them time when you factor in the benefits of the error prevention it delivers. But if they allow it to lapse, catching up becomes too overwhelming.

The only exception to this rule is data flow static analysis. Since this is actually an error-detection exercise, it's feasible to use it to clean up a large code base as a one-off act.

You've mentioned "policy" a few times now. What exactly do you mean by it?

A policy is an overarching mandate for how the organization expects code to be written and what quality practices they expect the team to follow (e.g., they expect all code to be peer reviewed before release or all pattern-based static analysis violations to be fixed within one day of their introduction).

Pattern-based static analysis must be treated as a policy. In other words, you need to configure all installations of the static analysis tool to check whether code is written in the expected way. Then, every time a violation is found, it needs to be fixed immediately because it's a violation of your policy. The tool's settings should be fine-tuned as needed to ensure that no false positives are reported. False positives de-sensitize developers to all violations—real and false—as a result of the "cry wolf" effect.

So do you find that policies are really being adhered to in the field?

When I'm called in to visit customers or potential clients, they often start off asking about tools and features. This is a red flag. When we start talking more, it usually comes to light that they previously acquired some static analysis tool, told the developers to use it, but didn't achieve the desired benefits because the developers never really started using it, or they eventually stopped using it.

At this point, I recommend they shift their focus from tool to process. Process and workflow is really what makes or breaks a static analysis initiative. You need a process that ensures static analysis is not only done continuously and consistently, but also done in a way that doesn't disrupt the normal workflow. You need to ingrain static analysis into your existing development process. To achieve this, the process has to be automated. The static analysis tool is instrumental in driving this process automation, but it's not the holy grail.

Case in point: Find a group where developers claim to be doing static analysis "as needed"—for instance, developers running an open source static analysis tool from their desktops. Ask them if any violations would be reported if you ran the static analysis tool on the entire code base. Most likely, yes. This means they are just paying lip service to static analysis. Without effective process integration and automation, static analysis efforts inevitably decay.

How do you go about automating the process?

An automated infrastructure is key. You want it working like a fine-tuned machine, so the following tasks are performed automatically in the background each night:

1. The source code repository is scanned to access the latest code and provide visibility into code base changes.
2. The most current code is analyzed by the static analysis tool.
3. Each violation is matched to the developer responsible for introducing it (using data gathered by scanning the source control system), and then distributed to that developer.

In the morning, each developer can go to his IDE, then import the violations found for code that he authored. To promote fast remediation, each reported violation links directly to the related source code.

I talk about automated infrastructure in detail in the book *Automated Defect Prevention: Best Practices in Software Management*. Chapter 1 is available for download at http://www.parasoft.com/adp book.

If an organization wants to get started with static analysis, what do you recommend as the first step?

Commit to making it a continuous part of your process, and start building a supporting infrastructure to drive that process. At minimum, you need an integrated automated infrastructure that performs the three tasks listed above. Parasoft Services can help you establish this; we have over 15 years of experience establishing sustainable static analysis processes that deliver greater productivity as well as significantly fewer software defects.

About Adam Kolawa

Adam Kolawa, Parasoft co-founder and CEO, is considered an authority on the topic of software development and the leading innovator in promoting proven methods for enabling a continuous process for software quality. In 2007, eWeek recognized him as one of the 100 Most Influential People in IT.

Kolawa has co-authored two books—*Automated Defect Prevention: Best Practices in Software Management* (Wiley-IEEE, 2007) and *Bulletproofing Web Applications* (Wiley, 2001)—and contributed a chapter to O'Reilly's *Beautiful Code* book. He has also written or contributed to hundreds of commentary pieces and technical articles for publications such as The Wall Street Journal, CIO, Computerworld, and Dr. Dobb's Journal, as well as authored numerous scientific papers on physics and parallel processing.

Kolawa holds a Ph.D. in theoretical physics from the California Institute of Technology. He has been granted 15 patents for software technologies he has invented.

About Parasoft Static Analysis Solutions

Parasoft solutions establish a continuous, automated process that centralizes management of pattern-based static analysis, data flow static analysis, and code metric calculation. They support not only Java, C/C++, and .NET languages (C#, VB.NET, and Managed C++), but also JavaScript, HTML, CSS, VBScript/ASP, XML, and WSDL (for SOA).

Parasoft's 15+ years of experience in unobtrusively integrating static analysis into existing development workflows is key to establishing it as a practical and sustainable process. Moreover, to ensure that the resulting process remains on track, reports provide managers visibility into policy adherence and alert them to areas where process improvement might be valuable.

To learn more about these solutions, contact Parasoft as described below, or visit http://www.parasoft.com/solutions.

About Parasoft

For 20 years, Parasoft has investigated how and why software errors are introduced into applications. Our solutions leverage this research to deliver quality as a continuous process throughout the SDLC. This promotes strong code foundations, solid functional components, and robust business processes. Whether you are delivering Service-Oriented Architectures (SOA), evolving legacy systems, or improving quality processes—draw on our expertise and award-winning products to increase productivity and the quality of your business applications. For more information visit: http://www.parasoft.com.

Contacting Parasoft

USA

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 (1) 64 89 26 00
UK: Tel: +44 (0)1923 858005
Germany: Tel: +49 89 4613323-0
Email: info-europe@parasoft.com

Asia

Tel: +886 2 6636-8090
Email: info-psa@parasoft.com

Other Locations

See http://www.parasoft.com/jsp/pr/contacts.jsp?itemId=268