
Ensure Quality Code Throughout
Your Software Development Process

# Static Analysis & Unit Testing for Medical Devices

July 2019

## Abstract

Regardless of industry, any company writing code understands the need for standardization and proof-checking, and the medical device sector is no different. What is different is the emphasis on safety, quality, performance, and security. While a lot of businesses have a standard practice they follow, Velentium is here to inform the industry on how static analysis, dynamic analysis, and more specifically, unit testing, can be developed and best applied to the development lifecycle for secure medical device software.

One of Velentium's Principal System Architect & Engineers, Satyajit "Sat" Ketkar, will explain how companies can adopt these best practices, as well as show in detailed steps the way Velentium accomplishes specific tasks internally using the software Parasoft. Sat has nineteen years of engineering experience, seven of them in medical device design. A majority of his career has revolved around electrical, firmware, software, and systems engineering, but recently he spent over eighteen months working for a European Union notified body. This experience allowed him to see product development in a different way, teaching him how to review and audit products for safety, quality, performance, and security.

# Contents



# Key Takeaways:

- What is Static Analysis, Dynamic Analysis, and Unit Testing?
- When and why should Static Analysis, Dynamic Analysis, and Unit Testing be a part of your development process?
- What standards should be addressed when performing Static Analysis?
- Step-by-Step instructions on how to implement Keil uVision and Parasoft together

# Background

Throughout this paper, we'll be taking a look at static analysis, dynamic analysis, and unit testing: what they are, how they developed, and how they are best applied to the secure development lifecycle for medical device software.

At the end of this paper, there is a link to a step-by-step guide aimed at configuring Parasoft for testing in medical device development, which is the tool we've identified as providing all the functionality we require at the best price point.

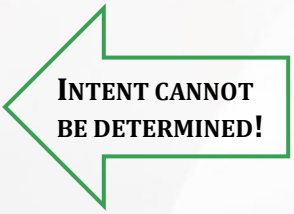
# Static Analysis

Static analysis, also called static code analysis, is a method of computer program review and debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure and helps to ensure that the code adheres to industry standards. By running the source code through a set of predetermined criteria, called checkers, static analysis tools expose various types of flaws in the code. Static analysis tools function independently of compilers, linkers, and hardware, and can analyze the most commonly utilized languages, C and C++.

# Limitations

Current Static analysis tools do not replace traditional code review entirely, but they make the code review process more efficient and productive. The next generation of static analysis tools do show promise of improved automation of the code review process, and with further development, this could become possible in the future.

Static analysis also cannot look for correctness based on the intent of the developer. For example, if you have a function that is called "multiply (A, B)" but when executed it returns A plus B, the analyzer cannot determine whether the developer's intent was to multiply or add A and B. Since this method of analysis does not execute the code or compare results with purpose, the possible bug would not be considered a violation or caught.




Another limitation with this method is that it cannot detect and enforce a developer's specific coding style unless the parameters of that style are manually defined and input before running the analysis. Even then, results may vary.

Static analysis occasionally reports false positives (saying there is a problem when there isn't one) and false negatives (assuming there isn't a problem when there is one). Fortunately, each generation of static analysis technology is continuing to decrease the occurrences of these flaws. In the 1970s, the false negative and positive rates were 25-40%. By the 1990s, these rates were in the 10% range; today, false positive and negative rates have improved to less than 1%.

# Historical Overview

The concept of static analysis was first introduced by Alan Turing in 1936 as he attempted to address the halting problem during his team's work on the Enigma machine.

Not much additional development occurred after Turing until the 1970s, when safety-critical firmware and software needed to be

interpreted and checked for programming errors, suspicious constructs, bugs, and stylistic errors before they were executed. Tools designed for this purpose were called Linters, and the most popular one was Lint (now PC-lint).

Static analysis tools have continuously evolved, and some, like PC-lint, are still being put to good use today. However, most of the activities that first-generation Linters performed have since been incorporated into today's compilers.

The 1990s brought several changes to static analysis. Accommodating code standards such as the then-new C89, beginning to address code quality and metrics, and decreasing the rates of false positives and negatives. One of the standard tools at the time was Coverity, which is less popular today, but was a pioneer in code quality analysis, checking issues like explicit casting and variable consistency.

In the early 2000s, the automotive, oil and gas, aerospace, medical, and other high-risk industries came together to establish standards for safe and secure code development. The third generation of static analysis began to emphasize safety, security, and higher standards for code quality. At the same time, static analysis concepts were adopted by large corporations. Microsoft, for example, incorporated some into their Intellisense IDs. Meanwhile, Lint, Coverity, and others released plug-ins for large-scale configuration management systems.

Static analysis tools have incorporated new standards based on C99 and further improved the false positive and false negative report rate to below 1%. One of the more popular tools currently is Parasoft, which not only facilitates compliance with industry standards but goes beyond those standards with a host of additional configurable checks available as well (which we'll discuss in-depth later).

## Guidelines & Standards

Coding rules and guidelines exist to ensure that software is:

- Safe – usable without causing harm. Especially critical in the medical field, where code that supports functions that would typically be innocuous could have a ripple effect impacting patient safety
- Secure – its vulnerabilities are mitigated
- Reliable – functions as it should, every time
- Testable – can be evaluated
- Maintainable – even as the codebase grows. We saw an example of unmaintainable code with the Toyota acceleration issue. When Toyota analyzed their dysfunctional code, they found that it had become unmanageably complex over years of continuous development with insufficient checks and oversight.
- Portable – functionality is the same across various environments and systems

Using established coding standards:

- Ensures compliance with ISO (the International Standards Organization), which is required for most major medical device markets
- Guarantees consistent code quality – no matter when or who writes the code
- Secures the software right from project start
- Reduces Costs by speeding up time-to-market through reduced variability and

uncertainty, as well as by decreasing overhead for post-market support

Following the regulations makes the code easy to review and debug, allowing for secure software to be developed more quickly and with greater assurance, increasing premarket development speed and decreasing the amount of patching necessary to fix issues or concerns post-market.

Even though IEC 62304, the ISO guidance on software development for medical devices, does not explicitly include static analysis as part of its direction, the FDA premarket guidance now specifically calls out secure static analysis testing. Other regulatory bodies such as those in the EU, Canada, and Australia are also catching up on requiring static analysis. Noncompliance exposes not only your business but also your potential patients to significant risk or harm.

Functional Standards define the minimum operational requirements of a system and its components. In the medical device industry, these are typically related to functional safety. Examples of these are shown below:

- IEC 61508 – "Functional safety of electrical/electronic/programmable electronic safety-related systems"
- IEC 62061 – "Safety of machinery: Functional safety of electrical, electronic and programmable electronic control systems"
- ISO/IEC TS 17961 – "Information Technology – programming languages, their environments, and system software interfaces"; security coding rules.

Coding Standards are a collection of coding rules, guidelines, and best practices to improve the safety, quality, and security of the implementation. These may also include guidelines on coding style. Examples of these are shown below:

- MISRA (Motor Industry Software Reliability Association) – one of the most popular coding standards. The original version (1998) emphasized safety, quality, and reliability, but it was revised in 2012 to include Static Analysis for Secure Testing (SAST) based on CERT C.
- CERT C by SEI (Software Engineering Institute) – created in 2008 specifically for security, using ISO 17961 as its baseline, and geared specifically for SAST. Its latest revision was published in 2016

CERT C and MISRA 2012 have some areas of overlap, but because of their different emphases, our best practice recommendation is to integrate CERT C with MISRA 2012 and apply them both.

In general, static analysis should be performed each time code is compiled and committed or pushed, prior to any formal code review, and before unit testing. It should become a habit to perform static analysis on a regular cadence throughout each project. The pattern then reinforces coding best practices, so that it becomes easier to remember the critical rules and apply them while developing. The result is not only a cleaner code base and a more reliable product but a faster, more efficient development cycle.

# Dynamic Analysis

Prior to the 1970s, little distinction was made between debugging and testing like there is today. If your code compiled, it was considered complete! However, in the 1980s, the aerospace industry introduced destructive or stress testing, which deliberately strains the component or system to expose faults and understand performance. Contrast this with nondestructive testing, which evaluates standard component or system behaviors in order to collect and understand performance metrics under normal conditions.

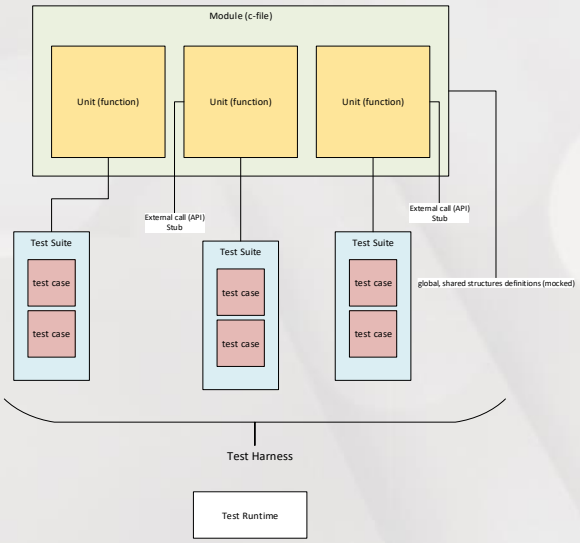
As developers began to distinguish between debugging and testing, it became a best practice for developers to debug software and then have a separate test group evaluate it. Later in the 1990s, developers decided it would be beneficial to integrate testing into the development cycle so that bugs could be prevented before the code reached the verification stage. This was the beginning of dynamic analysis as we know it.

Dynamic analysis includes assessments of *Modules*, such as an individual C-file (and can also incorporate its matching header file), and *Units*, or individual functions within a module. For testing purposes, both the API and the static units are counted separately.

- Unit Testing is white-box testing conducted on the smallest testable components of the software. In procedural languages like C, this is an individual function in isolation. In object-oriented programming, this may be an entire class or interface.
- Integration Testing (also called *module testing* or *interface testing*) is grey-box testing on related or grouped modules. These tests typically go through the API, purposely constraining themselves to the interface's functionality as it relates to the module or module grouping, and attempt to stress the interface. This includes low-level "fuzzing,"pushing garbage or noise as an input to see how the program responds.
- System Testing is black-box testing of the entire software, using only externally-available stimulus such as the Communication Interface Protocol (CIP). It involves both destructive/stress and nondestructive/evaluative testing and seeks to answers questions like, "What happens if we overload communications channels?" as well as, "How long does it take the program to return the answer to a complicated set of inputs?"

# Unit Testing

Unit testing consists of isolating individual units from each module and the overall software system and subjecting each group to a series of tests. External calls made by the unit are stubbed with mock functionality. Any shared parameters should also be mocked out because you want to test the individual unit as independently from the rest of the module as possible. Once isolated and mocked, each group within the module is challenged for various conditions with a pass/fail criteria.




Test Case: each test case exercises a specific functional or behavioral path within a unit (function) for a given module. Test cases should be completely independent of each other. If you have to combine test cases or suites in any manner, they are not being executed properly.

Test Suite: a collection of orthogonal tests cases for a given module

Mocks and Stubs: Mocks or stubs are specially generated functions that replace the actual function calls from the unit under test because they reside outside the module in scope. This gives the test developer flexibility to add/modify the stub functionality to inject the necessary stressed for a given test scenario or behavior. This also allows for removing hardware dependencies or requirements for testing.

Test Harness: a collection of test suites, stubs, and mocks along with test validation functionality

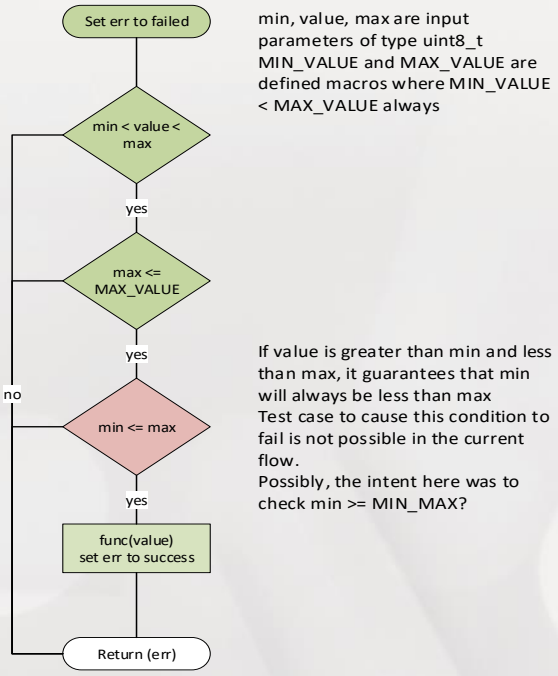
Test Runtime: a tool-specific run time executable or library

Unit testing provides statement coverage, meaning it assures that every line of code was executed and addresses every edge, branch, and condition. Next, it ensures that the code recognizes boundary conditions and correctly responds to an out-of-bounds input.

For example, if you typed in zero or six into a one-to-five input, the boundary conditions will detect the problem and enforce backup procedures. It detects security problems stemming from low-quality code, such as using unsafe string functionality. When static analysis identifies a possible security flaw, unit testing can confirm the validity of the static analysis. Unit testing assists with fault localization, isolating specific buggy code. It includes flow analysis, which checks permutations to see if there is unnecessary or repetitive code that can be merged or deleted to save space.

Finally, unit testing encompasses memory leak detection and concurrency defects. These last two are not always required for programs developed in embedded medical applications but do apply for programs designed to run on an operating system (PC or mobile platform).

Here's a simplified example of the kind of problem unit testing can uncover:




As written, no input value which passes the first two checks will ever fail the third. Writing a test to confirm that this check works correctly isn't possible, so testing has successfully identified a problem with the code (results which we would expect to fail the third check always pass).

Although it would be ideal to perform unit tests as often as possible, practical limitations prevent this from becoming a reality. That said, unit testing should always be performed:

(1) After completing each initial unit implementation (including coding/debugging)
(2) After static analysis, but before formal code review
(3) Every time the unit or its parent module is updated or refactored

Test-Driven Development (TDD) philosophy encourages unit testing and implementation to be done in parallel. Ideally, the tests would be generated toward the design prior to implementation – i.e., you should identify and devise testing during design review – but this is difficult in practice since it tends to push implementation milestones farther away from project start in the timeline. Look for a compromise between TDD best practice and project timeline.

Unit testing is required to comply with IEC 62304 as part of required unit verification for Class B and C medical devices. Note that 'unit verification' here does not necessarily mean that 100% unit testing coverage is required; risk analysis will determine how much unit testing is necessary, and which areas of the program can be assured through integration testing instead. Meanwhile, the FDA's updated premarket submission guidance does require unit testing with boundary conditions to be performed as part of SAST.