



# Embedded Systems Design

Researchers at the FDA's Office of Science and Engineering Laboratories investigating new techniques for analyzing software in medical devices are using static analysis tools to uncover potential flaws in a device under review.

feature

## Using static analysis to evaluate software in medical devices

BY RAOUL JETLEY AND PAUL ANDERSON

The Center for Devices and Radiological Health (CDRH) at the FDA is responsible for post-market surveillance of medical devices. If a device failure resulting in actual or potential serious injury or death is reported, the manufacturer of the medical device is required to investigate, determine the root cause and contributory factors, develop appropriate corrective actions, and report their findings to CDRH.

In cases where the adequacy of the manufacturer's investigation or corrective action is in question, CDRH may conduct an independent investigation. Commensurate with the threat to public health, CDRH can unilaterally take a range of actions, including issuing public health notifications or mandating a product recall.

Performing a post-market investigation, however, is not an easy task. This is particularly true in the case of software, where the execution is often user-driven and system-specific. To further complicate matters, device software is usually event-driven, resulting in failures that are often unpredictable and may not be easily reproducible. In such cases, the only way to trace the software flaws has historically been to manually review the source code itself. Given the complexity of modern medical-device software, this is a very difficult and time-consuming task for a third-party investigator with no prior knowledge of the software.

Recently the CDRH's Office of Science and Engineering Laboratories (OSEL) has been investigating the use of static analysis technology to assist in this task. This article gives a brief introduction to static analysis and explains how we used this technique to detect flaws.

### Listing 1 Example showing a NULL pointer dereference

```

24 unsigned int x;
25 char *p, *q
26
27 GetXValue (&x);
28 if (x < 100)
29 {
30     printf ("x is less than 100")
31     if (x == 10)
32         q= NULL
33     p = &x;
34     x = *q;
35 }

```

#### STATIC ANALYSIS

Historically, static analysis has been used mainly to enforce syntax checks and coding standards in software. Over the last few years, a new breed of static analysis tools, based on light-weight formal methods, has emerged that can be used to detect potentially fatal flaws in the software.

The flaws detected by these static analysis tools include run-time errors, such as buffer overruns, null pointer dereferences, race conditions, resource or memory leaks, and dangerous casts. Some advanced tools also incorporate a facility to detect inconsistencies in the code, such as redundant conditions or erroneous assumptions that may indicate programmer misunderstandings. Typically, when a potential flaw is found in the software, the tool generates a warning that allows the user to see not only where the flaw occurs, but the conditions that must hold in order for it to occur.

An advanced static analysis tool typically operates by performing an abstract or symbolic execution of the program. During this execution, program variables containing actual concrete values are replaced by corresponding symbolic values. The analysis proceeds by using these symbolic values to follow all possible paths through the code. Along each path, all possible symbolic values are recorded. As this execution proceeds, the analysis may learn facts about the variables and how they relate

to each other. It uses these facts to refine associated symbolic values and check for potential errors. If any of the values is determined to result in an error at any point along the path, a corresponding warning is issued.

As an example, consider the code snippet shown in Listing 1. When analyzing this code, the static analysis tool computes symbolic values for the vari-

### An advanced static analysis tool typically operates by performing an abstract or symbolic execution of the program.

ables  $x$ ,  $p$ , and  $q$  along all possible paths in the program. As it navigates through these paths, the tool learns facts about the variables and uses them to refine their corresponding symbolic values. Along the path comprising lines 29 through 35 for instance, the tool learns that  $x$  can only hold values between 0 and 100. Therefore, it refines the symbolic value associated with  $x$  to the range  $[0..100]$ . Further, it learns that when  $x$  is equal to 10, the pointer  $q$  is assigned the value NULL. Thus, the tool associates the symbolic value NULL with  $q$ , provided the predicate ( $x == 10$ ) is true.

Finally, on line 34, when  $q$  is dereferenced, the tool determines that a possible symbolic value for  $q$  may be NULL and issues a corresponding 'NULL Pointer Dereference' warning to indicate the same.

A common problem with static

analysis tools is the generation of false positives—warnings reported by the tool that are not genuine errors. False positives are generally caused due to a lack of domain-specific knowledge about the code. Typically, this involves variables that can only hold specific values and can never be assigned the erroneous symbolic values as computed through static analysis. A high false-positive rate is undesirable as it could result in needless effort for the user and may result in true positives (errors) being overlooked.

While completely eliminating false positives is not possible, most static analysis tools deal with this problem by providing users with configuration parameters to control the analysis. For example, the maximum number of paths explored per procedure may be specified to increase or limit the search-space for the tool. These configuration parameters help provide the tool with

the missing domain-specific knowledge required for accurate analysis, while allowing users to select the level of analysis most appropriate for their application.

#### CASE STUDY: POST-MARKET STATIC ANALYSIS

At OSEL, we used static analysis for the post-market review of a commercial medical device. The aim of this analysis was to determine all possible potential causes for failure in the software and to assess compliance to established software and quality control standards. GrammaTech's CodeSonar static analysis tool was employed to carry out the analysis. CodeSonar is a source-code analysis tool that performs a whole-program, interprocedural analysis on C/C++ code to identify complex programming bugs that can result in system crashes, memory corruption, and other serious problems.

The software for the device under review was implemented largely in C/C++, with some macros defined using low-level assembly code. The software was deployed in the form of three

## An uninitialized-variable warning generated by CodeSonar.

```
Line Source
someFile.c
Enter S_Keys
1252 static void S_Keys (void)
1253 {
1254     BUTTON    button;
1255     EVENT     button_press;
1256     BOOL      is_button_pressed;
1257     unsigned int number;
1258
1259     is_button_pressed = FALSE;
1260     number = 0;
1261     button_press = NULL;
1262
1263     while (number++ < NUMBER_OF_BUTTONS)
1264     {
1265         if (ProcessButton ((unsigned int*)&button) == FALSE)
1266         {
1267             Notify (button_press);
1268         }
1269     }
1270
1271     switch (button) /* Uninitialized Variable (ID: 10) */
1272
```

Figure 1

independent modules, comprising approximately 200,000 lines of code.

As the compiler used by the manufacturer was nonstandard, a mock en-

**The output of the analysis was generated as a navigable HTML report, with links to specific regions in the source code that contained the errors.**

vironment was created based on the documentation accompanying the source code. The result of this was a makefile that could be used to emulate the original compilation environment. In addition, several configuration parameters were changed from the default in order to improve the precision of the analysis. This included increasing the number of paths being searched in each procedure, maximizing the path finding effort, and setting the lower bound for the null pointer threshold to 1 to indicate that all address values (except NULL) could be dereferenced safely.

Finally, the tool was run for each of the three modules separately. The output of the analysis was generated as a navigable HTML report, with links to specific regions in the source code that contained the errors.

Figure 1 shows part of the output generated by the static analysis tool. The code in the figure is an “Uninitialized variable” warning. This warning is generated to report the attempted use of a variable that has not been initialized (names of the variables in the code have been deliberately changed to protect the manufacturer’s confidentiality).

The code shown in the figure depicts part of a procedure `S_Keys` that has a number of variables defined locally. Of these, the variables `is_button_pressed`, `number`, and `button_press` are initialized to default values at the start of the procedure. However, the variable `button` is not initialized. Instead, it is expected to be assigned a value by the function `ProcessButton`, to which it is passed as a parameter, invoked at line 1267.

Moreover, the function `ProcessButton` is invoked from within a conditional loop and may itself contain conditional statements that would not always guarantee a valid value being assigned to the variable `button`. Thus, when used in the `switch` statement on line 1272, `button` is flagged as an “Uninitialized variable.” This could lead

to an unintended path being executed along the `switch` statement, which may ultimately lead to device malfunction.

In all, static analysis of the code produced a total of 736 such warnings. A breakdown of these warnings classified by the different warning classes is listed in Table 1.

All of the reported warnings were inspected manually to determine if they constituted genuine problems. A number of warnings were discarded during this process, if they were determined never to be the direct cause of a device malfunction.

For example, the static code analyzer issues an “Unused value” warning when a variable is assigned a value that is never used. These types of warnings are typically harmless by themselves, but the tool reports them because some safety-critical coding conventions prohibit their use, and because they could be indicative of poor design and maintenance processes.

Warnings like these were discarded, unless they were suspicious for some other reason. A second group of warnings were discarded because they were false positives. As discussed in the previous section, false positives are impossible to avoid in general. However, some of these could be eliminated by choosing the configuration parameters based on domain knowledge.

As a result of this manual analysis, 127 of the 736 warnings reported were found to be of genuine concern; in other words, they either reflected poor quality control or had the potential to cause the device to malfunction. These warnings were submitted as part of a report to the CDRH compliance group to take further action as necessary.

The total effort expended during the post-market analysis was 210 person-hours. A majority of the effort was expended in configuring the build for the application and manual analysis of the results. While 210 person-hours is still a significant amount in terms of the effort required for the analysis, it is considerably less than what would have been required for a purely manual analysis. Additionally, the static analysis method

## Warnings reported by CodeSonar

Warning class	Warnings reported	Actual problems
Buffer overrun	6	
Buffer underrun	9	
Cast alters value	116	29
Ignored return value	14	
Division by zero	1	
Leak	25	
Missing return statement	13	1
Null pointer dereference	62	28
Redundant condition	139	4
Shift amount exceeds bit width	2	
Type overrun	3	
Type underrun	2	
Uninitialized variable	169	36
Unreachable code	34	20
Unused value	23	
Useless assignment	118	9

Table 1

provides for a much more reliable means for tracing errors in the software as opposed to the manual process.

**PREVENTIVE MEDICINE**

Static analysis is a valuable tool for post-market investigation. By reasoning about potential run-time errors in the software, static analysis provides an independent, standardized, and repeatable inspection of a medical device's software, as part of a broader scientific analysis of the device. Further, providing the precise location of the failure and a corresponding execution trace enables the investigator to trace the root cause of

failure to its origin in the source code.

This ability not only helps reduce time and effort involved in post-market investigation, but also leads to a more accurate means for post-market analysis, as opposed to manual inspection. Most importantly, the use of static analysis allows the post-market investigator to evaluate the product, in this case the software, and not just the processes involved in developing it.

Much as static analysis helps the investigator, it can be leveraged to even greater effect by medical-device manufacturers. The manufacturers can use static analysis to help find flaws early in

the development cycle. Static analysis lends itself readily to verification and validation activities and can easily be incorporated as part of the manufacturers' software-development processes. Doing so facilitates a deeper assessment of the code before releasing it to market and helps establish conformance to good programming practices.

On the basis of this experiment, we have reason to believe that static analysis—whether used in pre-deployment analysis by the manufacturer or during post-market surveillance by an investigator—has the potential to greatly reduce software anomalies and lead to safer, more dependable medical devices.

■   
 Raoul Jetley is a researcher at the US FDA, Center for Devices and Radiological Health/Office of Science and Engineering Laboratories. His research interests include formal methods and static analysis of medical device software. Jetley received his PhD in computer science from North Carolina State University. Contact him at [raoul.jetley@fda.hhs.gov](mailto:raoul.jetley@fda.hhs.gov).

Paul Anderson is vice president of engineering at GammaTech, a spin-off of Cornell University that specializes in static analysis. He received his B.Sc. from Kings College, University of London and his Ph.D. in computer science from City University London. Paul manages GammaTech's engineering team and is the architect of the company's static analysis tools. A significant portion of his work has involved applying program analysis to improve security. Paul can be reached at [paul@grammatech.com](mailto:paul@grammatech.com).