Static Code Analysis and Dynamic Testing: Complementary Techniques for your Software Quality

Increased recall campaigns, delayed deliveries, difficulties in delivering the promised functions on time: software quality is not evident. The development of good software is only possible through consistent action, adherence to standards and the use of mature test and quality assurance tools. Bad software leads to monetary losses and deterioration of the corporate image. Embedded software is even more critical, as it is mostly used in safety-critical applications. Here, software errors can endanger human lives and must therefore be avoided at all costs.

For this reason, standards like ISO 26262, IEC 61508 or DO178-C have strict requirements regarding the quality of development and testing of software.

To ensure quality, both static code analysis procedures and the testing of executable software during dynamic analysis (including unit tests) are necessary. As each of the two approaches uncovers only a part of the existing defects, both complementary methods are necessary.

Static Analysis Tools can already be used at the beginning for the Development

While dynamic analysis requires the code to be executed, static analysis does not. Static analysis tools can therefore be used early in the development process during the implementation phase. For this reason, static code analysis contributes massively to the success of the project - the earlier errors are found, the more cost-effective are the fixes.

Without writing test cases, static code analysis tools check the code for syntax, semantics, control flow and data flow anomalies, concurrency problems as, well as for programming rules. Many bugs and security vulnerabilities are uncovered.

It is recommended, that the code is analyzed statically on a regular basis right from the start of development - preferably by the individual developer before he checks-in his code. It makes sense to only submit the code to a further verification step such as code reviews, unit tests or integration tests when the static code analysis no longer indicates any errors. With this procedure, the number of error messages during the final inspection before delivery can be dramatically reduced.

Static analysis tools are particularly useful in the development of embedded systems, where languages such as C and C++ are used. These languages give developers a lot of freedom - unfortunately also for writing faulty source code. Bugs like Null-Pointer-Exceptions, Buffer-Overflows or problems with global variables are common. Such errors can be avoided with static code analysis.

Dynamic Tests are also necessary

As soon as the software can be executed, static analysis should be supplemented by dynamic tests.

Dynamic tests are mainly used to prove the functional correctness of a system. Usually, they are performed as soon as the first code components are executable. An important part of these tests is the code coverage analysis, which ensures that all (important) parts of the code will be tested.

Code coverage tools like Testwell CTC++ from Verifysoft (1) place counters at all relevant points in the source code (instrumentation of the code), to measure if this code parts have been executed during the test runs. As embedded systems have usual only limited memory space, it is important that the overhead of this instrumentation remains small. In addition, code coverage tools should only have a minimal impact on performance in order to avoid malfunctions in time-critical systems. Coverage tools are usually integrated into the development environments – the code instrumentation is automatic. After the test runs, the coverage analyzer generates reports that allow to see in detail which functions were executed, and which were not.

For safety-critical software code coverage is mandatory. The standards DO-178C (aviation), ISO 26262 (automotive), EN 50128 (railway), and the general standard IEC 61508 stipulate high code coverage levels up to Modified Condition Decision Coverage (MC/DC) to demonstrate testing of all conditions or decisions in a software.

To achieve good Quality, both Static Analysis and Dynamic Tests are needed

To guarantee high quality, a combination of static analysis, sufficient testing during execution of the software (dynamic tests) associated with code coverage is necessary. Our real-life example of a project of a household appliance shows why both methods need to be used and why only employing one testing or analysis technique may lead to fatal consequences.

The manufacturer developed a software for the control unit of a washing machine product line. It was written in C and was supposed to run on a microcontroller. This identical control unit together with the identical software was meant to be used in all machines of this product line; thereby, specific functionalities would be turned on or off respective to the machine type. More expensive machines, for example, were equipped with a sensor, measuring the “staining degree” of the laundry, and enabling the program to customize the duration of the main wash cycle. The washing times of more simple machines without sensors were to remain constant.

```
size_t durationMainWashCycle(size_t prog, size_t load, size_t staining) {
	if (((prog == 3) || (prog == 5) || (prog == 7)) && (load < 5)) {
		return staining * 5;
	}
	else if (((prog == 4) || (prog == 6)) && (load < 5)) {
		return staining * 8;
	}
	else if (((prog == 4) || (prog == 6)) && (load < 3)) {
		return staining * 7;
	}
	else {
		return staining * 9;
	}
}
```

Fig. 1 *Function for the calculation of the duration of the main washing cycle*

Testcases to ensure sufficient software quality were developed under stipulation of a MC/DC code coverage of 100%. The manufacturer believed static analysis to be expendable.

One needs to look at the source code (Fig. 1) to describe the resulting issues.

The function in Fig. 1 calculates the duration of the washing cycle depending on the selected washing program as well as load size and “staining degree” of the laundry. Related to this function, the test cases listed in the table below were executed during module testing, whereby the “staining degree” is a factor assessed during the tests. To which degree the “product version” influences this result will be discussed later.

test case no	product_version	prog	load	staining	result	expected result
1	11	3	4	3	15	15
2	11	5	4	3	15	15
3	11	7	4	3	15	15
4	11	3	6	3	27	27
5	12	4	2	1	**8**	**7**
6	12	6	2	1	**8**	**7**
7	13	4	4	1	8	8
8	13	6	4	1	8	8

Fig. 2 *Executed test cases related to function „durationMainWashCycle()“*

However, test coverage of the function considered was only at 71%. The report on test coverage (Fig. 3) together with the testing result immediately uncovered an issue: The programme path with the if-condition in the beginning of line 31

```
else if (((prog == 4) || (prog == 6)) && (load < 3)) {
              return staining * 7;
```

was not passed through, although the respective test cases (No 5 and 6) were executed. Moreover, the actual testing result for these test cases differed from the expected results. The test coverage report illustrates that the if-condition from line 28 was passed through instead.

An exchange of these two else if-conditions in the code eradicated the error. A new test run now resulted in test coverage of 95% with matching actual and expected test results.

The report on test coverage showed that an additional test case was needed to reach 100% test coverage.

test case no	product_version	prog	load	staining	result	expected result
9	14	6	6	1	9	9

Hits/True	False	Line	Source
8		24	size_t durationMainWashCycle(size_t prog, size_t load, size_t staining) {
3	5	25	if (((prog == 3) || (prog == 5) || (prog == 7)) && (load < 5)) {
1		25	1: ((T) || (_) || (_)) && (T)
1		25	2: ((F) || (T) || (_)) && (T)
1		25	3: ((F) || (F) || (T)) && (T)
	1	25	4: ((T) || (_) || (_)) && (F)
	0	25	5: ((F) || (T) || (_)) && (F)
	0	25	6: ((F) || (F) || (T)) && (F)
	4	25	7: ((F) || (F) || (F)) && (_)
+		25	MC/DC (cond 1): 1 + 7
+		25	MC/DC (cond 2): 2 + 7
+		25	MC/DC (cond 3): 3 + 7
+		25	MC/DC (cond 4): 1 + 4, 2 - 5, 3 - 6
3		26	return staining * 5;
		27	}
4	1	28	else if (((prog == 4) || (prog == 6)) && (load < 5)) {
2		28	1: ((T) || (_)) && (T)
2		28	2: ((F) || (T)) && (T)
	0	28	3: ((T) || (_)) && (F)
	0	28	4: ((F) || (T)) && (F)
	1	28	5: ((F) || (F)) && (_)
+		28	MC/DC (cond 1): 1 + 5
+		28	MC/DC (cond 2): 2 + 5
-		28	MC/DC (cond 3): 1 - 3, 2 - 4
4		29	return staining * 8;
		30	}
0	1	31	else if (((prog == 4) || (prog == 6)) && (load < 3)) {
0		31	1: ((T) || (_)) && (T)
0		31	2: ((F) || (T)) && (T)
	0	31	3: ((T) || (_)) && (F)
	0	31	4: ((F) || (T)) && (F)
	1	31	5: ((F) || (F)) && (_)
-		31	MC/DC (cond 1): 1 - 5
-		31	MC/DC (cond 2): 2 - 5
-		31	MC/DC (cond 3): 1 - 3, 2 - 4
0		32	return staining * 7;
		33	}
		34	else {
1		35	return staining * 9;
		36	}
		37	}

Fig.3 *Code Coverage Report (MC/DC Coverage) of Testwell CTC++ Test Coverage Analyzer*

With the missing test case, the stipulated test coverage of 100% was reached.

After successfully concluding integration testing and solving some minor issues, the washing machines went into production and were delivered. After some time, there were complaints from unhappy customers. Some machines had an insufficient washing result; this was due to the premature termination of the washing cycle. There were also reports of machines elongating the main washing cycle for several hours. Replication of this behavior proved not to be possible.

Initially, a malfunction of the staining sensor was suspected. Therefore, they were exchanged within the warranty. However, it became clear quite quickly that this did not resolve the issue.

After closer inspection of the complaint cases, it became apparent that they were limited to a specific machine type in the product line. Hence, the possibility of a software error was considered, and an external service provider was tasked to run a static code analysis.

```
size_t getStainingLevel(size_t product_version) {
	size_t y;
	if (product_version < 12) { //products w/o staining sensor
		y = 3;
	}
	else if (product_version > 12) {  //products with staining sensor
		y = readStainingSensor();
	}
	return y;
}
```

Fig. 4 *Determination of the staining degree depending on the product version.*

The stipulation that all models from model no 12 interrogate the staining sensor and all other models work under a constant staining value has been executed incorrectly. The variable "y" for model 12 in the function „`getStainingLevel()`" remains uninitialized and passes an undefined value for the staining degree. Since the value was inconspicuous during module testing, this error remained undetected.

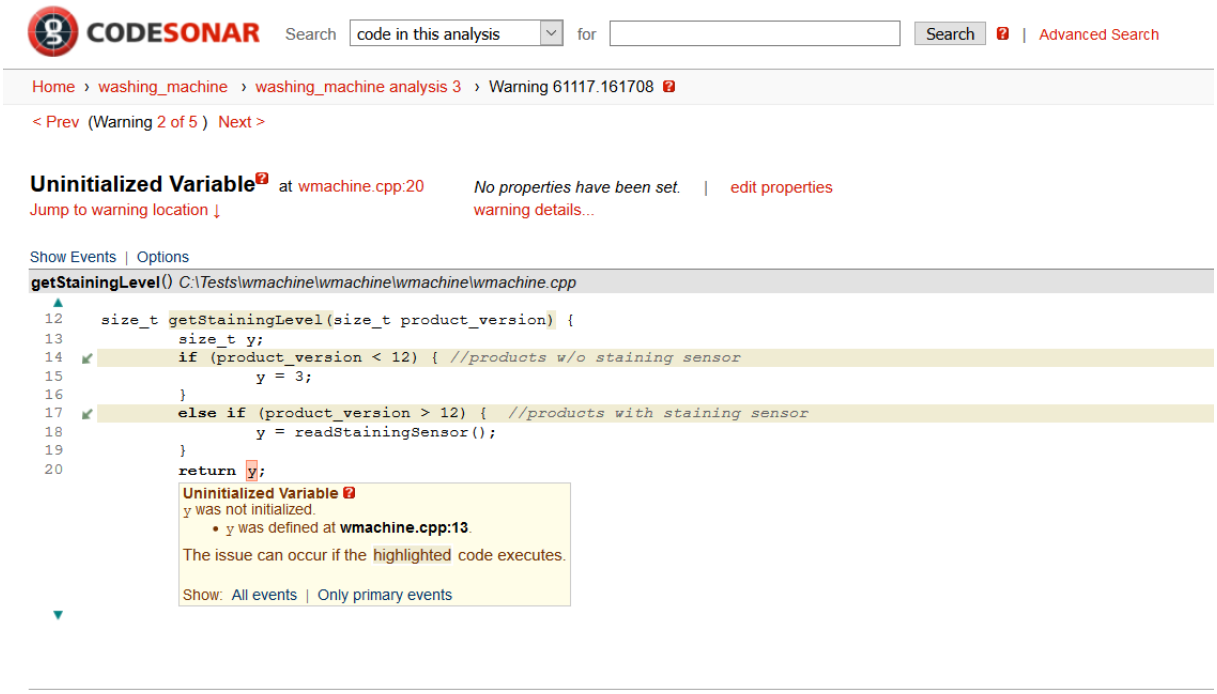

Fig. 5 *Uninitialized variable causes undefined behavior* shown by the static analysis tools GrammaTech CodeSonar (2)

Moreover, the result of the static source code analysis of the simplified code (Fig. 6) illustrates that the previously described error of the unreachable code section could have most probably already been discovered early on during the implementation process.

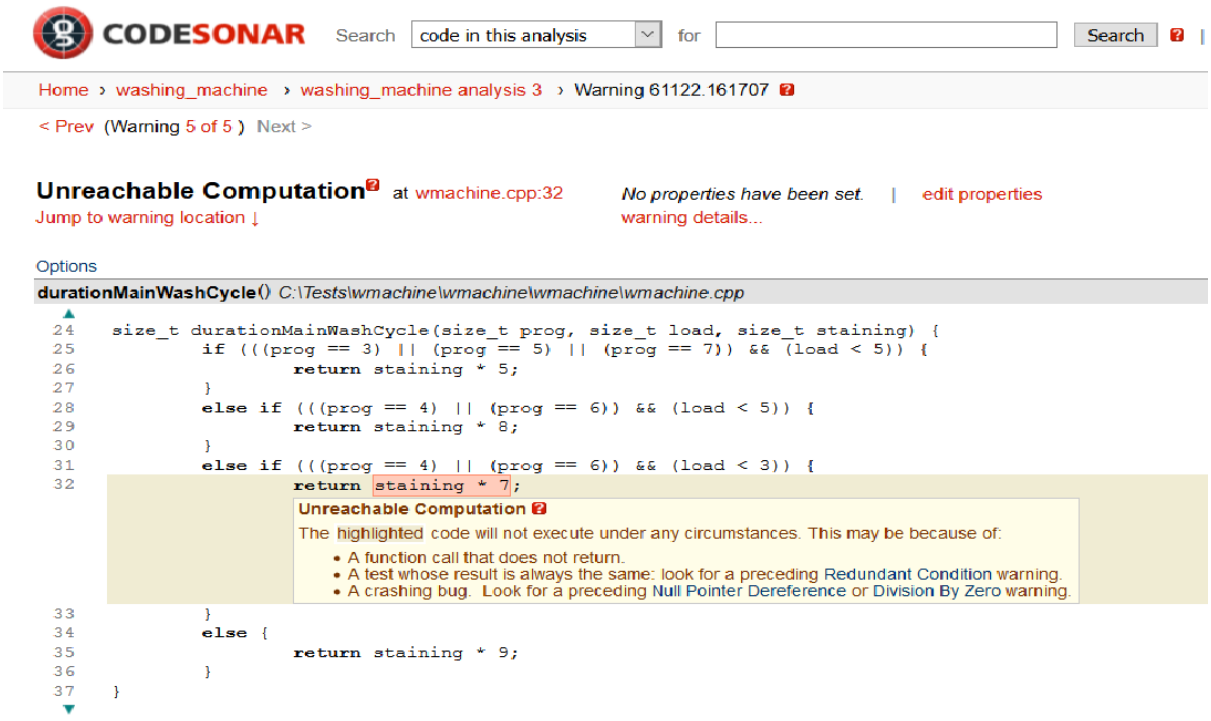

Fig. 6 *Unreachable code shown by GrammaTech CodeSonar*

The error was only discovered by static and dynamic analysis together. Unfortunately, static code analysis was only implemented retroactively by the washing machine manufacturer. In the development stage, error correction would have been more economical.

Timely static analysis together with dynamic testing would have avoided the costly product recall and the related image loss.

The electric appliance manufacturer now employs both static analysis and dynamic testing for test coverage for all its software projects.

Autor

Royd Lüdtke is Director for static analysis tools at Verifysoft Technology, a company dedicated to software quality with 700+ customers in more than 40 countries. Royd Lüdtke has extensive experience as an application engineer and consultant, holds several patents and is the author of reference books.

Links

(1) www.verifysoft.com

(2) www.grammatech.com

