Error-free software with static code analysis and dynamic testing

In order to guarantee high software-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 shows why both methods need to be used to ensure high software quality and why only employing one testing or analysis technique may lead to fatal consequences.

The real-life example of a project of a household appliance manufacturer meaningfully illustrates this statement.

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 with ARM technology. 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.

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 in order to describe the resulting issues. The original code cannot be published due to legal reasons and would be too complex in this context. The code pictured here in Fig. 1 is heavily simplified and focuses only on defining the problem.

```
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*

The function in Fig. 1 calculates the duration of the washing cycle depending on the selected washing programme 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()“*

Fig. 3 shows the result of the reading of the test coverage.

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	}

Abb.3 *reading of the test coverage (MC/DC)*

However, test coverage of the function considered was only at 71%. The report on test coverage 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

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 behaviour 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.

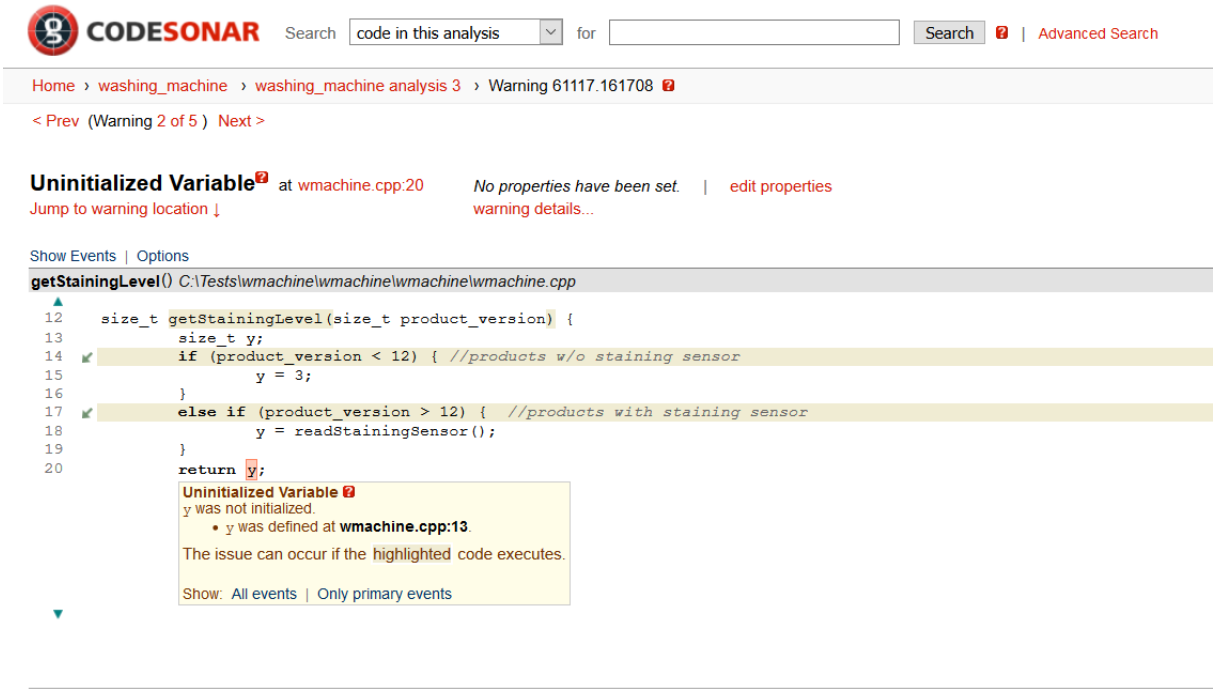

Fig. 5 *Uninitialized variable causes undefined behaviour*

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.

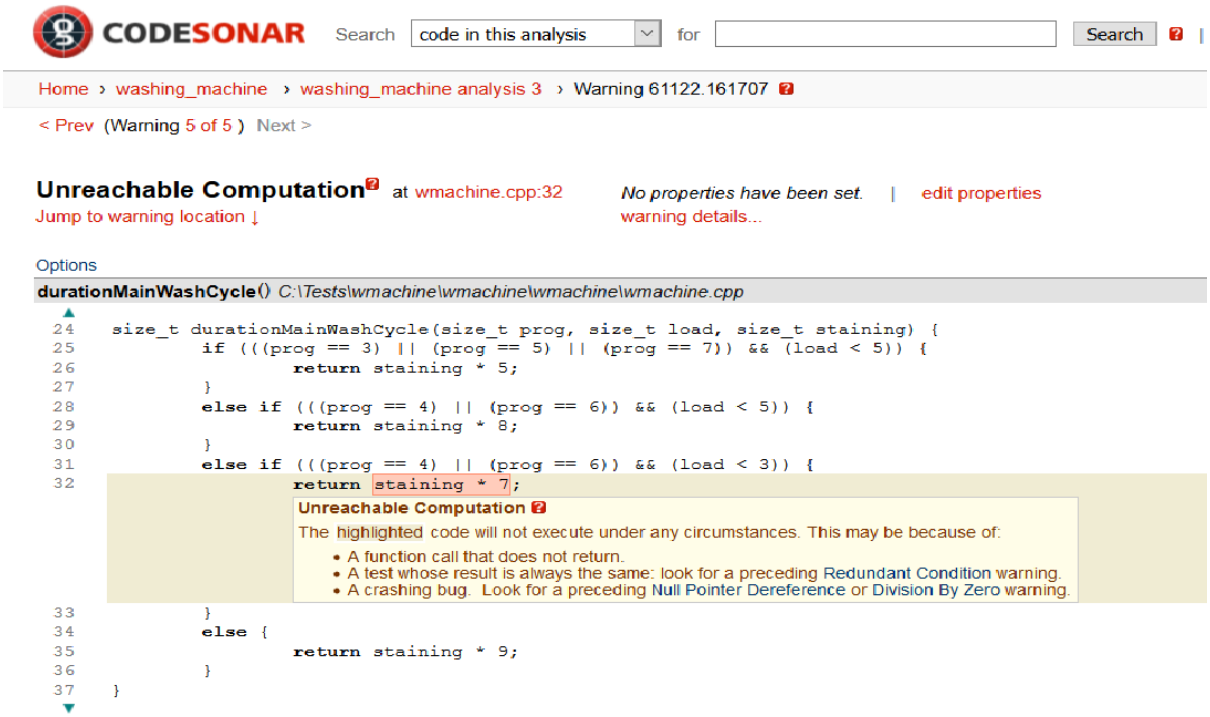

Fig. 6 *Unreachable code*

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.

Further information:

The tool CodeSonar[i] by GrammaTech was used for static code analysis. The tool Testwell CTC++[ii] by Verifysoft was used to measure test coverage.

The French translation of this article was published in the standard work “Pratique des Tests Logiciels” by Jean-François Pradat-Peyre and Jacques Printz at the publishing house Dunod[iii]

Authours:

Royd Lüdtke is Director for Static Code Analysis Tools at Verifysoft Technology GmbH.

Roland Person is technical customer advisor and mainly looks after dynamic code analysis and test coverage.

[i] https://www.verifysoft.com/de_grammatech_codesonar.html
[ii] https://www.verifysoft.com/de_ctcpp.html
[iii] ISBN 978-2-10-081995-9