Gemeinsam zum Erfolg:

Statische Codeanalyse und dynamische Tests

Softwareentwicklern stehen zwei Techniken zum Testen von Software zur Verfügung: dynamische Tests und die statische Codeanalyse. Erst wenn beide Methoden kombiniert werden, lassen sich Fehler in der Software weitgehend ausschließen.

Vermehrte Rückrufaktionen, verzögerte Auslieferungen, Schwierigkeiten, die versprochenen Funktionen rechtzeitig auszuliefern: Softwarequalität ist nichts, was vom Himmel fällt. Gute Software entsteht nur durch konsequentes Handeln, Einhalten von Normen und den Einsatz ausgereifter und funktionsreicher Qualitätssicherungstools. Schlechte Software führt zu monetären Verlusten und Imageschäden. Besonders kritisch ist Software für Embedded-Systeme, da diese oft in sicherheitskritischen Anwendungen eingesetzt wird. Hier können Softwarefehler Menschenleben gefährden und müssen daher auf jeden Fall vermieden werden.

Branchennormen wie die ISO 26262, IEC 61508 oder DO178-C schreiben deshalb strikte Anforderungen hinsichtlich der Qualität von Entwicklung und Test von Software vor. Um die Qualität sicherzustellen, werden zwei Methoden unterschieden: statische Codeanalyseverfahren und das Testen lauffähiger Software während der dynamischen Analyse – unter anderem mit Unit-Tests. Beide Verfahren sind komplementär – jeder der beiden Ansätze deckt verfahrensbedingt jeweils nur einen Teil der vorhandenen Fehler auf. Größte Fehlerfreiheit kann erst im Zusammenspiel von statischer und dynamischer Analyse erreicht werden.

Einsatz statischer Analysetools früh im Entwicklungszyklus

Muss bei der dynamischen Analyse der Code ausgeführt werden, so ist dies bei der statischen Analyse nicht der Fall. Daher können statische Analysetools schon früh im Entwicklungsprozess in der Implementierungsphase genutzt werden und tragen deshalb massiv zum Projekterfolg bei – denn je früher getestet wird, desto günstiger ist die Fehlerbehebung.

Statische Codeanalysetools überprüfen den Code auf Syntax, Semantik, Kontrollfluss- und Datenflussanomalien, Nebenläufigkeitsprobleme und Einhaltung von Codier-Richtlinien/Standards. Ohne das Schreiben von Testfällen wird eine Vielzahl an Bugs und sicherheitsgefährdender Schwachstellen frühzeitig aufgedeckt.

Idealerweise wird der Code bereits vom Beginn der Entwicklung an regelmäßig – am besten bereits durch den einzelnen Entwickler vor dem Einchecken und dann mit zunehmender Integration von Modulen – statisch analysiert. Hierbei ist es sinnvoll, den Code erst dann einem weiteren Verifikationsschritt wie Code-Reviews, Unit-Tests oder Integrationstests zuzuführen, wenn die statische Codeanalyse keine Fehler mehr anzeigt. Bei diesem Vorgehen kann die Anzahl der Fehlermeldungen bei der Abschlussprüfung vor Auslieferung entscheidend reduziert werden.

Der Einsatz von statischen Analysetools ist vor allem bei der Entwicklung von Steuerungs- und Embedded-Systemen sehr sinnvoll. Da hier oft sehr hardwarenah programmiert wird, kommen Sprachen wie C und C++ zum Einsatz, die den Entwicklern viele Freiheiten geben – leider auch beim Schreiben von fehlerhaftem Quellcode. So fügt ein C/C++-Compiler keinen Code ein, der testet, ob angeforderter Speicher auch zugeteilt wurde oder eine zu kopierende Zeichenkette in das Zielarray passt. Dies passiert beispielsweise im folgenden Programm, in welchem mit der Funktion »malloc()« durch das Betriebssystem Speicher angefordert wird:

```
int hallofehlerwelt ()
{   char *p;
    p = (char *)malloc (20);
    strcpy(p, “Hallo Fehlerwelt“);
    printf(“%s\n“,p);
}
```

Der Fehler liegt hier darin, dass davon ausgegangen wird, dass die 20 angeforderten Bytes auch bereitgestellt wurden und der Zeiger »p« auf ein Feld mit zwanzig Zeichen zeigt. Schlug die Speicheranforderung fehl, schreibt die String-Copy-Funktion »strcpy()« in die Speicherstelle 0, was in der Regel zum Programmabsturz führt, weil jedes moderne Betriebssystem diese Speicherstelle schützt. Die Funktion »strcpy()« prüft nicht, ob an der Stelle p genügend Platz für den String vorhanden ist, den es dann kopiert. Fügt man also noch ein »schöne« zwischen »hallo« und »fehlerwelt« ein, schreibt strcpy mehr als zwanzig Zeichen in den Speicher und damit in Bereiche, die andere Daten oder Programmcode enthalten können. Das Resultat ist ein Speicherüberlauf (Buffer-Overflow).

Die Verwendung globaler Variablen kann ebenfalls fehlerträchtig sein, weil Compiler nicht prüfen, ob auf eine globale Variable lesend zugegriffen wurde, bevor sie initialisiert wurde. Zudem lassen sich Variablen-Typen in C und C++ relativ leicht verändern, was zu impliziten Wertveränderungen führen kann. Die korrekte Vorgehensweise zu prüfen, wird umso schwieriger, wenn viele Entwickler an unterschiedlichen Modulen gleichzeitig arbeiten, die voneinander abhängig sind und niemand mehr alle Programmentwicklungen im Blick hat.

Solche Fehler können durch die statische Codeanalyse vermieden werden. Sie erledigt die Arbeit, die sich gängige Compiler sparen und erzeugt bei einem Analyselauf Abbilder der komplexen Abläufe und Datenzugriffe. Das Werkzeug für die statische Analyse durchläuft den Source-Code wie ein Compiler, erzeugt aber statt Code ein Analyseabbild, das vom Tool umfangreich ausgewertet wird.

Besonders fortschrittlich ist hierfür das Werkzeug CodeSonar von GrammaTech. CodeSonar erstellt aufgrund des Quellcodes Modelle der Datenströme und Zugriffe, analysiert diese und listet alle potenziellen Schwachstellen auf. Es wird angezeigt, wo Funktionen und globale Variablen verwendet werden. Über eine grafische Ausgabe sind
die Zusammenhänge leicht erkennbar. Jedes einzelne Problem wird mit ausführlichen Fehlerbeschreibungen angezeigt.

Gute Werkzeuge für die statische Analyse bieten im Vergleich zu einfachen Analysewerkzeugen einen weitaus größeren Komfort, indem sie Fehlermeldungen nach Kritikalität sortieren. Das Entwicklerteam kann sich somit zunächst der Behebung der

schwerwiegenden Fehler widmen und weniger kritische Meldungen dann bearbeiten, wenn hierzu noch Zeit bleibt. Warnungen, die vom Team als nicht relevant angesehen werden, können sogar ganz unterdrückt werden.

Der Einsatz von »Advanced Static Analysis Tools« führt daher bei Entwicklern und Managern zu höherer Akzeptanz. Die etwas höheren Kosten für die Anschaffung des Tools werden in der Regel schnell wieder hereingeholt.

Dynamische Tests ergänzen die statische Codeanalyse

Sobald die Software im fortgeschrittenen Projektstadium ablauffähig ist, sollte die statische Analyse durch dynamische Tests ergänzt werden. Sie dienen vor allem dazu, die funktionale Korrektheit eines Systems nachzuweisen.

Üblicherweise werden dynamische Tests ausgeführt, sobald erste Bestandteile des Codes lauffähig sind, und begleiten dann die Entwicklung bis zum fertigen Produkt. Ein wichtiger Bestandteil dieser Tests ist die Code-Coverage-Analyse. Hiermit wird sichergestellt, dass jeglicher Code einer Anwendung auch durchlaufen – d. h. getestet – wurde. Programmteile, die offensichtlich für einen gewissen Zweck entwickelt wurden und damit auch eine Funktion haben sollten, die aber nie aufgerufen werden, lassen vermuten, dass ein Programmier- oder Verfahrensfehler vorliegt. Im einfachsten Fall ist es Code, den niemand gelöscht hat, nachdem er nicht mehr gebraucht wurde, und der »nur« Speicherplatz belegt. Im schlimmsten Fall sind es Initialisierungs- oder andere Funktionen, die selten zum Einsatz kommen, im Bedarfsfall aber wichtige Aufgaben haben.

Code Coverage Analyser wie Testwell CTC++ von Verifysoft ergänzen zur Messung der Testabdeckung die Programme durch passende Zähler, die an allen relevanten Stellen im Source-Code platziert werden – Instrumentierung des Codes – und analysieren, ob der entsprechende Codeteil durchlaufen wurde. Wichtig bei solchen Tools ist, dass die Instrumentierung möglichst wenig Speicherplatz belegt, da in Embedded-Systemen häufig nur limitierter Hauptspeicher zur Verfügung steht. Außerdem dürfen solche Tools die Leistungsfähigkeit nur minimal beeinflussen, um in zeitkritischen Regelungssystemen keine Fehlfunktionen zu generieren, die sonst nicht auftreten würden. In der Regel sind Coverage-Tools in die Entwicklungsumgebungen integriert, sodass kein manuelles Platzieren von Testcode nötig ist. Nach den Testläufen erzeugt der Coverage-Analyser Berichte, über die Entwickler sich bis ins Detail anschauen können, welche Funktionen ausgeführt wurden und welche nicht. Ein nützlicher Nebeneffekt der Code-Coverage-Analyse ist die Angabe, wie häufig einzelnen Codeteile aufgerufen wurden.

Bei sicherheitskritischer Software sind dynamische Testverfahren und der Nachweis der Code-Coverage aus gutem Grund verpflichtend. So fordern die DO-178C in der Luftfahrt, die ISO 26262 im Automobilbereich, die EN 50128 im Schienenverkehr sowie die allgemeine Norm IEC 61508 für Software mit hoher Kritikalität hohe Code-Coverage-Stufen wie die Modified Condition Decision Coverage (MC/DC), um den Test aller Bedingungen bzw. Entscheidungen in einer Software nachzuweisen.

Dies ist beispielsweise für If-Then-Abfragen mit mehreren logischen Und- oder Oder-Verknüpfungen wichtig. So wird bei einer Abfrage wie:

```
if (sensorActivity == TRUE || sensorB > grenzwert(…))
{ … }
```

die Funktion »grenzwert()« im zweiten Teil der Abfrage hinter dem logischen Oder »||« nicht ausgeführt, wenn sensorActivity den Wert True hat. Das ist auch sinnvoll, weil unabhängig vom Ergebnis des zweiten Vergleichs der Gesamtausdruck True sein wird, wenn sensorActivity schon True ist. Ist die Funktion »grenzwert()« auch für die Initialisierung von globalen Variablen oder I/O-Registern verantwortlich, kann die Nichtausführung zu schwer auffindbaren Fehlern führen, weil die Entwickler fälschlicherweise davon ausgehen, dass diese Funktion bei dieser Abfrage immer ausgeführt wird.

Oft sind logische Abfragen noch wesentlich verschachtelter – bei logischen Und-Abfragen mit False-Prüfungen gilt das Gleiche. Auf den ersten Blick ist nicht erkenntlich, ob hier ein Fall vorliegt, bei dem Codeteile nie ausgeführt werden.

Kombination von statischer Codeanalyse und dynamischem Testen

Um eine gute Softwarequalität zu gewährleisten, ist eine Kombination von statischer Analyse und dynamischen Tests in Verbindung mit einer Messung der Testabdeckung erforderlich. Im folgenden Praxisbeispiel wird deutlich, warum beide Verfahren zur Absicherung der Softwarequalität genutzt werden müssen und warum die Konzentration auf nur eine Test- oder Analyseform gefährliche Konsequenzen haben kann.

Für die Steuereinheit einer Waschmaschinen-Produktreihe wurde eine Software in der Programmiersprache C für einen Mikrocontroller mit ARM-Prozessorkern geschrieben. Für alle Waschmaschinen der Produktreihe wurde eine identische Steuereinheit mit identischer Software genutzt, wobei je nach Maschinentyp bestimme Funktionen ein- bzw. abgeschaltet wurden. Die Maschinen der etwas höheren Preisklasse sollten z. B. mit einem Sensor ausgestattet werden, um den Verschmutzungsgrad der Wäsche zu messen und dem Programm so zu ermöglichen, die notwendigen Zeiten des Hauptwaschgangs automatisch und bedarfsgerecht anzupassen. Die Waschzeiten der einfacheren Maschinen ohne Sensor sollten dagegen konstant sein.

```
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;
        }
}
```

Listing 1. Beispiel Waschmaschinen Software: Funktion zur Berechnung der Waschdauer des Hauptwaschgangs.

Zur Sicherstellung einer hinreichenden Softwarequalität wurden Testfälle unter der Vorgabe einer MC/DC-Testabdeckung von 100 % erstellt. Auf eine statische Analyse wurde zunächst verzichtet.

Die Waschdauer des Hauptwaschgangs wurde mit dem Code in Listing 1 berechnet. Die Funktion errechnet die Waschdauer in Abhängigkeit vom gewählten Waschprogramm, der Beladungsmenge und dem Verschmutzungsgrad der Wäsche.

Testfall Nr.	Nr. product_ version	prog	load	staining	Ergebnis	erw. Ergebnis
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

Bezogen auf diese Funktion wurden während der Modultests die in der Tabelle 1 aufgeführten Testfälle ausgeführt, wobei der Verschmutzungsgrad »staining« in das Testergebnis als Faktor einfließt. Inwieweit der Maschinentyp »product_version« das Ergebnis beeinflusst, wird später erläutert.

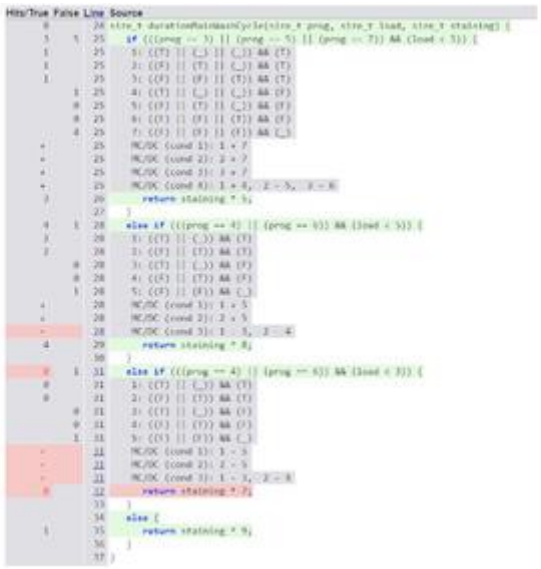

Bild 1. Messung der Testabdeckung (MC/DC) mit dem Code Coverage Analyser Testwell CTC++.

Das Ergebnis der Messung der Testabdeckung zeigt Bild 1. Die Testabdeckung der betrachteten Funktion lag zunächst nur bei 71 %. Aus dem Report der Testabdeckung in Verbindung mit den Testergebnissen wurde sofort ein Problem sichtbar: Der Programmpfad mit der if-Bedingung mit Beginn in Zeile 31 (siehe Bild 1) – in Listing 2 isoliert dargestellt – wurde nicht durchlaufen, obwohl die entsprechenden Testfälle (Nr. 5 und 6) ausgeführt wurden.

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

Listing 2. Die im Test als problematisch erkannte Programmzeile in der Funktion zur Berechnung der Waschdauer des Hauptwaschgangs.

Das Testergebnis für diese Testfälle wich zudem von den erwarteten Ergebnissen ab. Aus dem Testabdeckungsbericht wird deutlich, dass stattdessen die if-Bedingung ab Zeile 28 durchlaufen wurde.

Ein Tausch der beiden else-if-Bedingungen im Code behob diesen Fehler. Ein erneuter Testlauf ergab nunmehr eine Testabdeckung von 95 %, wobei die Testergebnisse den Erwartungen entsprachen.

Aus dem Report der Testabdeckung war ersichtlich, dass für eine Testabdeckung von 100 % noch ein zusätzlicher Testfall benötigt wird (Tabelle 2).

	Nr.					
Testfall Nr.	**product_ version**	**prog**	**load**	**staining**	**Ergebnis**	**erw. Ergebnis**
9	14	6	6	1	9	9

Ein Test unter Berücksichtigung des zuvor fehlenden 9. Testfalls ergab dann die geforderte Testabdeckung von 100 %.

Nach dem erfolgreichen Abschluss der Integrationstests gingen die Waschmaschinen schließlich in Produktion. Einige Zeit nach der Auslieferung kam es jedoch zu Reklamationen: bei einigen Maschinen war das Waschergebnis unzureichend, da der Hauptwaschgang bereits nach kurzer Zeit beendet war.

```
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;
}
```

Listing 3. Ermittlung des Verschmutzungsgrads in Abhängigkeit vom Typ der Waschmaschine (product_version).

Es wurden auch Fälle berichtet, bei denen die Maschinen die Hauptwäsche auf mehrere Stunden ausgedehnt hatten. Zuverlässig nachstellen ließ sich das Verhalten nicht. Die Reklamationen waren allerdings auf einen bestimmten Maschinentyp der Produktreihe beschränkt, sodass die Möglichkeit eines Softwarefehlers in Betracht gezogen und eine statische Codeanalyse durchgeführt wurde.

Bei der statischen Codeanalyse fiel auf, dass die Vorgabe, bei allen Maschinentypen ab dem Modell Nr. 12 den Verschmutzungssensor abzufragen und alle Modelle darunter mit einem konstanten Verschmutzungswert arbeiten zu lassen, fehlerhaft umgesetzt wurde (Listing 3).

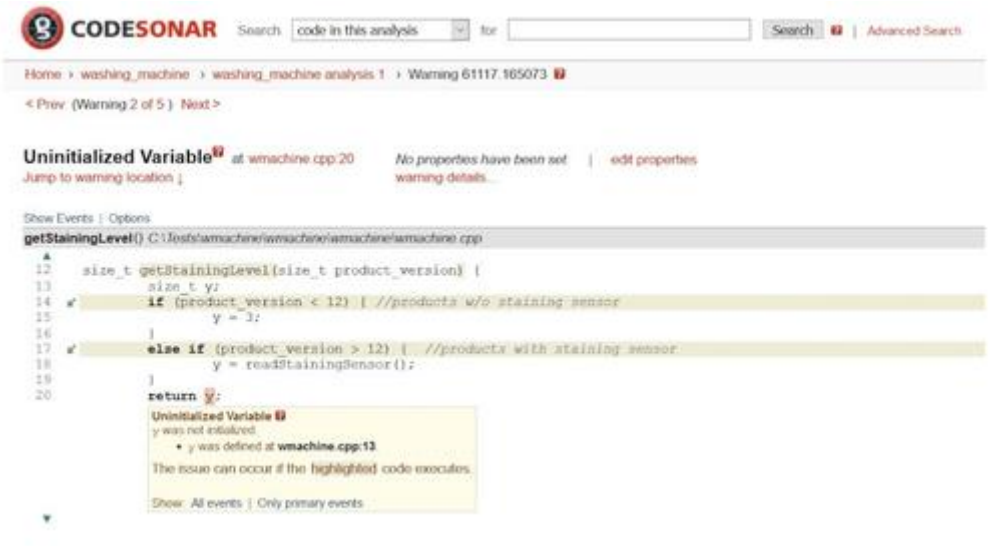

Bild 2. Eine nicht initialisierte Variable im Code (siehe Listing 3) – gefunden per statischer Codeanalyse – verursacht unbestimmtes Verhalten.

Für das Modell Nr. 12 bleibt die Variable »y« in der Funktion »getStainingLevel()« uninitialisiert und übergibt damit einen unbestimmten Wert für den Verschmutzungsgrad. Da während der Modultests der Wert nicht auffällig war, blieb dieser schwerwiegende Fehler unentdeckt (Bild 2).

Das Ergebnis der statischen Quellcodeanalyse (Bild 3) zeigt, dass der Fehler des nicht erreichbaren Codeabschnitts mit statischer Codeanalyse bereits zur Implementierungszeit hätte aufgedeckt werden können. Beide Fehler konnten nur durch die Kombination von statischer und dynamischer Analyse aufgedeckt werden.

Bild 3. Mit der statischen Codeanalyse lässt sich nicht erreichbarer Code finden.

Statische Codeanalyse und dynamische Tests kombiniert mit der Messung der Code-Coverage müssen komplementär eingesetzt werden, um möglichst viele Fehler ausschließen zu können. Die statische Codeanalyse kann dabei bereits früh im Entwicklungsprozess

eingesetzt werden und somit hohe Kosten einsparen. Richtig eingesetzt, bieten Tools zur statischen und dynamischen Codeanalyse erhebliches Potenzial zur Kosteneinsparung und zur Steigerung der Produktivität.

Der Autor

Royd Lüdtke

leitet bei Verifysoft Technology den Bereich Pre-Sales und Support für Statische Codeanalysetools. Er hat umfangreiche Erfahrung als Applikationsingenieur und Berater, hält mehrere Patente und ist Autor von Veröffentlichungen im IT-Bereich.

luedtke@verifysoft.com