Qualität von Software ist kein Zufall

Gute Software-Qualität kann nur im Zusammenspiel von statischer Analyse und dynamischen Tests erreicht werden.

Gute Software erfüllt die erwartete Funktionalität, ist sicher, zuverlässig und gut wartbar. Es ist wichtig, dass Software möglichst keine Fehler hat, damit sowohl die Funktionale Sicherheit (engl. Safety), als auch die Angriffssicherheit (engl. Security) sichergestellt ist. Hierzu kommen während der Softwareentwicklung idealerweise zwei unterschiedliche Verfahren zum Einsatz, die komplementär sind: die statische Code-Analyse und das Testen zur Laufzeit, auch dynamische Analyse genannt. Beide Methoden decken verfahrensbedingt nur einen Teil der vorhandenen Probleme auf. Erst wenn sie gemeinsam genutzt werden, kann die Qualität der Software entscheidend gesteigert werden.

Welcher Software-Projektmanager oder Entwickler kennt das nicht: verzögerte Auslieferungen, Überschreitung von Budgets, hoher Zeit- und Kostenaufwand für Fehlerkorrekturen, sowie Schwierigkeiten, die versprochenen Funktionen korrekt umzusetzen. Software-Qualität ist kein Selbstläufer, sondern nur durch konsequentes Handeln, Einhalten von Normen und den Einsatz ausgereifter Test- und Analysewerkzeuge erreichbar.

Gute Softwarequalität trägt zur Reputation einer Firma bei und hilft mittel- und langfristig entscheidend dabei, Kosten zu sparen. Bei der Entwicklung sicherheitskritischer Software sind Vorgehensweisen und Tests durch Normen vorgeschrieben, um eine Gefährdung von Gesundheit und Menschenleben durch mangelhafte Software auszuschließen. Um die Qualität sicherzustellen gibt es zwei komplementäre Ansätze, die erst im Zusammenspiel die größtmögliche Fehlerfreiheit sicherstellen können: Statische Code-Analyse und das Testen lauffähiger Software in Zusammenhang mit dem Nachweis einer ausreichenden Testabdeckung bzw. Code Coverage.

Statische Code-Analyse findet Fehler sehr früh

Insbesondere bei der Entwicklung von Embedded-Systemen kommen wegen der hardwarenahen Programmierung oft Sprachen wie C und C++ zum Einsatz, die hochperformanten und kompakten Code ermöglichen. C und C++ lassen Entwicklern viele Freiheiten bei der Programmierung – und leider auch beim Schreiben von fehlerhaftem Quellcode.

So überprüft ein C/C++-Compiler nicht, ob angeforderter Speicher auch zugeteilt wurde oder eine zu kopierende Zeichenkette in das Zielarray passt. Fehler wie Null-Pointer-Exceptions oder Buffer-Overflows können die Folge sein. Auch durch nicht initialisierte Variablen kann unbestimmtes und gefährliches Programmverhalten verursacht werden. Variablen-Typen, die sich in C und C++ relativ leicht verändern lassen, können zu impliziten Wertveränderungen führen. Die korrekte Entwicklung zu prüfen, wird schwierig, wenn viele Entwickler an unterschiedlichen Modulen voneinander abhängigen Modulen gleichzeitig arbeiten, aber niemand das „Große Ganze“ der Programmentwicklungen im Blick hat.

Werkzeuge für die statische Code-Analyse helfen hier entscheidend. In einem Analyse-Lauf werden Abbilder der komplexen Abläufe und Datenzugriffe

Source file: C:\Projects\hcontrol\regulators.c
Instrumentation mode: multicondition
TER: 71 % (20/28) structural, 71 % (17/24) statement
To files: Previous | Next

TER % - multicondition	TER % - statement	Calls	Line	Function
75 % - (6/8)	83 % - (5/6)	14	4	lights()
100 % (2/2)	100 % (1/1)	4	20	close_windows()
100 % (2/2)	100 % (1/1)	8	25	open_windows()
100 % (2/2)	100 % (3/3)	4	30	open_windows_for()
100 % (2/2)	100 % (1/1)	4	37	heat()
0 % - (0/2)	0 % - (0/1)	0	42	air_condition()
60 % - (6/10)	55 % - (6/11)	8	47	temperature_control()
71 % - (20/28)	**71 %** - (17/24)			**regulators.c**

Autor:
Klaus Lambertz,
Gründer und Geschäftsführer
Verifysoft Technology GmbH
www.verifysoft.com

True	False	Line	Source
14		4	void lights(enum light_status goal)
		5	{
0	14	6	if (goal == off)
		7	{
		8	printf("Light is switched off.\n");
		9	}
8	6	10	else if (goal == on)
		11	{
		12	printf("Light is switched on.\n");
		13	}
6	0	14	else if (goal == dimmed)
		15	{
		16	printf("Lights are dimmed.\n");
		17	}
14		18	}

erstellt, die anschließend umfangreich ausgewertet werden. Tools für die statische Analyse erstellen basierend auf dem Quellcode Modelle der Datenströme und Zugriffe, analysieren diese und listen alle potenziellen Schwachstellen auf. Schnell wird klar, wo Funktionen und globale Variablen verwendet werden. Grafische Ausgaben zeigen die Zusammenhänge in der Software. Probleme werden mit einer ausführlichen Fehlerbeschreibung gemeldet und können dadurch schnell behoben werden.

Programmierrichtlinien für saubere Programmierung

Wegen der Schwachstellen hardwarenaher Sprachen wie C und C++, ist man früh daran gegangen, Vorgaben zu entwickeln, die die Flexibilität dieser Sprachen zugunsten größerer Sicherheit beschränken und klare Vorgaben machen, welche Sprachelemente eingesetzt werden dürfen und welche nicht. Hierzu gehören Coding Guides wie MISRA C und MISRA C++, aber auch Build Security In (BSI) des US Departments of Homeland Security, die Power-of-Ten-Rules der NASA und die JPL-Regeln des amerikanischen Jet Propulsion Laboratories, die auf die Power-of-Ten und MISRA-C Regeln aufsetzen, um Risiken der Nutzung von multi-threaded Software zu vermeiden.

Gute Tools für die statische Codeanalyse tragen zur Überprüfung der Einhaltung dieser Regeln bei. Die Tools lassen sich in alle gängigen Entwicklungsumgebungen integrieren und können im Rahmen von Continuous-Integration eingesetzt werden.

Werkzeuge für die statische Codeanalyse tragen also einerseits zur Entwicklung von regelkonformer Software bei und decken andererseits auch Programmierfehler auf, die sich trotz der Beachtung von Coding Standards in den Code „eingeschlichen“ haben.

Das perfekte Duo

Dynamische Tests überprüfen die funktionale Korrektheit der Software – Code Coverage Tools sichern die Vollständigkeit der Tests. Ergänzend zur statischen Codeanalyse müssen dynamische Tests – also Tests zur Laufzeit – durchgeführt werden, um die funktionale Korrektheit des Systems nachzuweisen. Sobald erste Code-Bestandteile lauffähig vorliegen, soll mit dem dynamischen Testen begonnen werden. Prinzipiell gilt hierbei: je schwerwiegender die Folgen von Fehlern sein können, desto umfassender muss getestet werden. Um festzustellen, welche Teile des Quellcodes getestet worden sind, nutzt man Code Coverage Analyser. Diese Werkzeuge zur Messung der Testabdeckung platzieren an allen relevanten Stellen des Quellcodes Zähler (Code-Instrumentierung), die protokollieren ob bzw. wie oft der entsprechende Codeteil getestet wurde. Beim Test von embedded Software ist es wichtig, dass diese Zähler möglichst wenig Speicherplatz einnehmen und die Performance nur minimal beeinflussen, um in zeitkritischer Software keine Fehlfunktionen zu generieren. Nach den Testläufen erzeugt das Code Coverage Tool eine Übersicht, die im Detail zeigt, welche Funktionen ausgeführt wurden und welche nicht.

Coverage-Tools auch für kritische Software

Bei sehr kritischer Software ist es erforderlich, alle Ausprägungen von Mehrfachbedingungen (Multi-Conditions) abzutesten. Normen wie die ISO 26262 im Automotive-Bereich und die DO178-C in der Luftfahrt schreiben hierfür die Modified Condition/Decision Coverage (MC/DC) vor. Gute Coverage-Tools unterstützen dieses Coverage-Niveau und geben gezielte Hinweise darauf, warum bestimmte Funktionen nicht aufgerufen worden sind. Hiermit wird für den Tester ersichtlich, welche Kombinationen der jeweiligen Bedingungen noch getestet werden müssen.

Statische Analysen und dynamische Tests

Beide Ansätze, die statische Quellcode-Analyse und dynamische Tests das Testen von Software inkl. der Messung der Testabdeckung mit einem Code-Coverage-Tool ergänzen sich ideal. Erst beide Verfahren zusammen sind in der Lage, nahezu alle Fehlerquellen aufzuspüren. Darüber hinaus liefern die Werkzeuge auch noch wichtige Erkenntnisse, wie die verschiedenen Code-Teile miteinander agieren, und tragen somit dazu bei, die Komplexität im Hinblick auf eine bessere Wartbarkeit zu senken. Eingebunden in gängige Software-Entwicklungsumgebungen sind sie integraler Bestandteil des Software-Entwicklungs-Live-Cycle und unterstützen Programmierer in jeder Phase der Software-Entwicklung. ◄

CTC++ Coverage Report - Execution Profile #1/3

Directory Summary | Files Summary | Functions Summary | Untested Code | Execution Profile
To files: First | Previous | Next | Last | Index | No Index

Source file: calc.c
Instrumentation mode: multicondition **Reduced to:** MC/DC coverage
TER: 81 % (13/16) structural, 91 % (10/11) statement

Hits/True	False	Line	Source
		1	/* File calc.c ---------- */
		2	#include "calc.h"
		3	/* Tell if the argument is a prime (ret 1) or not (ret 0) */
Top			
9		4	int is_prime(unsigned val)
		5	{
		6	unsigned divisor;
		7	
2	7	8	if (val == 1 || val == 2 || val == 3)
1		8	1: T || _ || _
0		8	2: F || T || _
1		8	3: F || F || T
	7	8	4: F || F || F
		8	MC/DC (cond 1): 1 + 4
		8	MC/DC (cond 2): 2 - 4
		8	MC/DC (cond 3): 3 + 4
2		9	return 1;
5	2	10	if (val % 2 == 0)
5		11	return 0;
58	2	12	for (divisor = 3; divisor < val / 2; divisor += 2)
		13	{
0	58	14	if (val % divisor == 0)
0		15	return 0;
		16	}
2		17	return 1;
		18	}

***TER 81% (13/16) of FILE calc.c
91% (10/11) statement

Directory Summary | Files Summary | Functions Summary | Untested Code | Execution Profile
To files: First | Previous | Next | Last | Top | Index | No Index