Mit Hyper-Coverage zur sicherheitszertifizierten Software

Embedded-Software-Entwicklung

© Adobe Stock / THAWEERAT

Autor:
Michael Wittner
Geschäftsführer
Razorcat Development GmbH
info@razorcat.com
www.razorcat.com

Für eine Functional-Safety-Zertifizierung müssen Entwickler die Code-Coverage für den gesamten Quellcode inklusive aller Code-Varianten nachweisen. Doch wie lässt sich nicht getesteter Code in den originalen C/C++-Quelldateien erkennen? Eine Lösung bietet hier die Ermittlung einer „Hyper-Coverage".

Testabdeckung

Als „Code-Coverage" (Testabdeckung) bezeichnet man den Nachweis, dass alle Teile eines C/C++-Quellcodes möglichst vollständig getestet wurden. Die Normen empfehlen dabei je nach Einstufung einer sicherheitskritischen Anwendung passende Coverage-Maße wie beispielsweise Statement/Branch-Coverage (C0/C1) oder MC/DC-Coverage. Wurden alle verfügbaren Tests durchgeführt, erhält man eine Aussage, ob und welche Teile des Codes nicht getestet wurden.

Die Coverage-Messung stützt sich dabei auf den Kontrollfluss der Funktionen/Methoden einer Quelldatei und prüft die erreichten Programmzweige bzw. Bedingungen. Stellt sich die Frage, welcher Code tatsächlich im Kontrollfluss abgebildet wird. Die vielfachen Möglichkeiten des Präprozessors erlauben es, durch den Einsatz von Makros aus ein und derselben Quelldatei mehrere unterschiedliche Programme zu erstellen.

Code-Varianten

Aus dem Source-Code einer Quelldatei werden durch den Präprozessor zunächst alle Makros aufgelöst, sodass ein präprozessierter Code entsteht, der letztendlich vom Compiler übersetzt wird. In diesen Prozess fließen die folgenden Informationen ein:

- Die Quelldatei selbst bildet die Grundlage für die Analyse, in der Teile des originalen Source-Codes über Makros ausgeblendet sein können.
- Eingebundene Header-Dateien können über Makros kontrollierte Konfigurationen enthalten, die wiederum andere Teile des Source-Codes sichtbar machen oder weitere Makros definieren.
- Wichtig sind auch die Compiler-Optionen für die jeweilige Quelldatei, wie sie z. B. über eine Entwicklungsumgebung oder ein Makefile eingesteuert und auf der Kommandozeile an den Compiler übergeben werden.

Aus diesen Teilen entsteht der präprozessierte Code, der jeweils eine bestimmte Variante des originalen Source-Codes repräsentiert. Die Präprozessor-Makros werden dabei entweder durch #define / #ifdef-Anweisungen in den Header-Dateien oder durch Compiler-Switches aus der Konfiguration beim Aufruf des Compilers übergeben. Dieser präprozessierte Code stellt auch die technische Basis für Testwerkzeuge dar, auf dem die Analyse und Instrumentierung für die Coverage-Messung durchgeführt wird.

Varianten testen

Beim Test von Varianten ist es entscheidend zu wissen, welche Code-Zeilen aus der Original-Quelldatei in der jeweiligen Variante tatsächlich existieren. Und auch umgekehrt: Gibt es Code in der Origi-

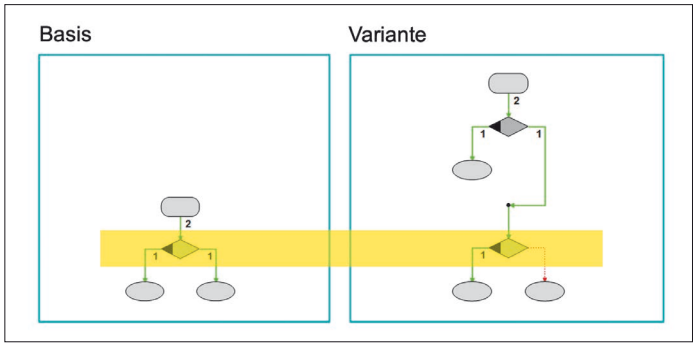

Bild 1: Unterschiedlicher Kontrollfluss zweier Varianten

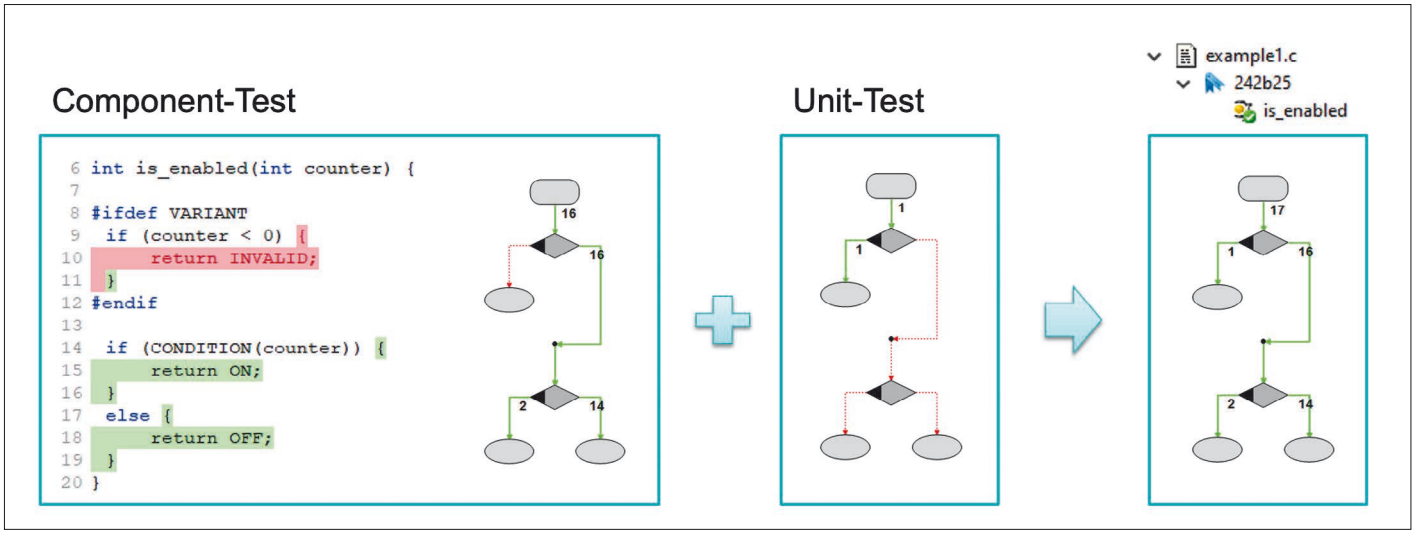

Bild 2: Kombination der Coverage aus Unit- und Integrationstest

nal-Quelldatei, der in keiner Variante enthalten ist?

Zunächst muss jede existierende Code-Zeile einer Variante auch genau in dieser Variante getestet werden. Für jede einzelne Variante muss eine möglichst vollständige Code-Coverage erreicht werden. Über Varianten von Testfällen lassen sich die meist relativ ähnlichen Code-Strukturen der Varianten gut testen, so dass der Testaufwand reduziert werden kann. In Bild 1 ist der Kontrollfluss von zwei Varianten mit der erreichten Code-Coverage dargestellt.

Man sieht dort, dass die Coverage-Ergebnisse aufgrund der unterschiedlichen Kontrollflüsse nicht einfach übertragen werden können, aber die Testfälle aus der Basis-Implementierung genutzt werden könnten, um den in der Variante fehlenden Programmzweig auf der rechten Seite noch abzudecken. Eine Vererbung und Anpassung der Basis-Testfälle für die Variante spart Aufwand bei der Testerstellung und verhindert redundante Testfälle.

Zusammenführen von Coverage

Beim Test mit vielen Varianten stellt sich die Frage, ob sich die Coverage-Ergebnisse zusammenfassen lassen, um den Testaufwand zu reduzieren. Diese Möglichkeit bietet sich zunächst nur für unterschiedliche Tests derselben Variante: Bei identischer Präprozessor-Datei ist die Code-Struktur ebenfalls identisch. Damit können die Coverage-Ergebnisse unproblematisch über den Kontrollfluss zusammengeführt werden.

Beispielsweise lässt sich fehlende Coverage für Statements/Branches oder Bedingungskombinationen einer Funktion durch zusätzliche Unit- und Integrations-Tests ergänzen. In Bild 2 wird auf der linken Seite die erreichte Coverage aus dem Integrationstest-Test dargestellt. Der rot unterlegte Programmzweig mit der Sonderbehandlung in dieser Funktion wird im Integrationstest nicht erreicht und daher durch einen weiteren Unit-Test ergänzt. In Summe ergibt sich damit eine vollständige Coverage für diese Funktion in dieser Code-Variante (rechts im Bild 2).

Source Files / Test Objects	C1	MC/DC
example2.c		
3d5787		
func2	100 %	66.66 %
func3	100 %	100 %
bdc232		
func1		
func2	50 %	
func3	100 %	

Bild 3: Coverage-Übersicht pro Funktion der jeweiligen Varianten

Hierarchie aus der Source-Code-Analyse

Aus der Analyse des präprozessierten Source-Codes der Varianten ergibt sich eine Hierarchie von Quelldateien mit deren Präprozessor-Varianten und den jeweils darin enthaltenen Funktionen. Bild 3 zeigt eine Quelldatei mit zwei Varianten und den Coverage-Ergebnissen der enthaltenen Funktionen. In dieser Übersicht wurde die kombinierte Coverage aus verschiedenen Tests bereits pro Funktion zusammengefasst.

Die Übersicht zeigt zwei noch offene Probleme: Obwohl die Funktionen aus derselben Quelldatei hervorgehen, kann die Coverage aufgrund unterschiedlicher Kontrollflüsse nicht kombiniert werden. Zum anderen fehlt eine Coverage-Aussage auf der Ebene der Original-Quelldatei.

Hyper-Coverage als Lösung

Eine Zusammenfassung der Coverage-Ergebnisse unterschiedlicher Varianten ist nur auf Zeilenbasis der ursprünglichen Quelldatei möglich. Aus der Analyse des Source-Codes jeder Variante kann ermittelt werden, welche Zeilen der ursprünglichen Quelldatei zu einem Statement/Branch oder zu einer Bedingung in dieser Variante gehören. Die Zeilen können daher wie folgt bewertet werden:

- Wenn die Coverage-Ergebnisse einer Variante erfolgreich sind, dann können die dazugehörigen Zeilen der Quelldatei ebenfalls als erfolgreich markiert werden.
- Falls mindestens eine Coverage-Art in einer Variante eine fehlende Abdeckung ergibt, dann wird die Zeile als nicht abgedeckt markiert.
- Falls es in keiner Variante Coverage-Ergebnisse zu einer Zeile gibt, dann wird diese Zeile ebenfalls als nicht abgedeckt markiert.

Auf diese Art können nicht getestete Code-Zeilen einfach entdeckt werden. In Bild 4 ist das Ergebnis der Hyper-Coverage auf Source-Code-Ebene dargestellt. Der gelb markierte Teil zeigt einen Code-Bereich, für den entweder keine Tests vorhanden sind oder die Tests nicht ausgeführt wurden. Die rot markierten Teile zeigen fehlende Coverage aus unterschiedlichen Coverage-Arten. Für die unten aufgelistete Funktion „func3“ wurden genügend Tests durchgeführt, so dass die geforderte Code-Coverage erreicht wurde.

Vorteil der Hyper-Coverage

Der Vorteil der Hyper-Coverage besteht darin, dass zwar eine zeilenbasierte Auswertung vorgenommen wird, aber die zugrundeliegenden Coverage-Ergebnisse aus der gewählten Coverage-Art vollständig berücksichtigt werden. Wenn beispielsweise MC/DC-Coverage gefordert ist, dann muss für eine Zeile mit einer komplexen Bedingung in allen Varianten eine vollständige MC/DC-Coverage erreicht

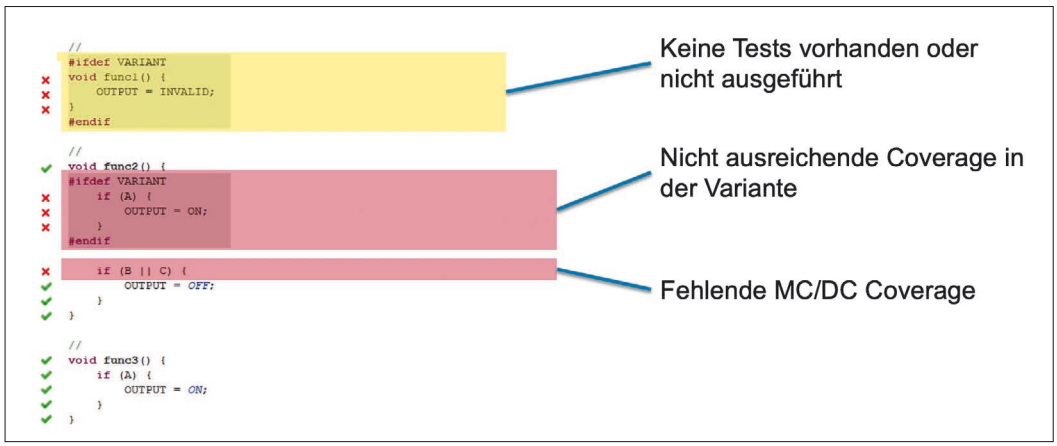

Bild 4: Zeilenweise Bewertung mit der Hyper-Coverage

werden, damit die Zeile als erfolgreich bewertet gilt.

Regressionstests

Die Ergebnisse aus einer erweiterten Analyse der Quelldateien können neben der Nutzung der Coverage-Ergebnisse auch zur Optimierung des Regressionstests benutzt werden. Ein optimierter Regressionstest von Software spart Zeit und Ressourcen und ermöglicht kontinuierliches Testen im Entwicklungsprozess. Wichtig ist die korrekte Erfassung von Abhängigkeiten, so dass nur die relevanten, aber vor allem auch die unbedingt notwendigen Tests wiederholt werden.

Code-Änderungen nur aufgrund von dateibasierten Prüfungen der Quell- und Header-Dateien zu erkennen, ist in der Regel zu grob, weil z. B. die Änderungen eines globalen Headers dazu führen würden, dass alle Tests wiederholt werden müssten.

Ein Beispiel

Aufgrund der oben geschilderten Source-Code-Analyse stehen deutlich genauere Informationen zur Erkennung von Änderungen zur Verfügung. Jede einzelne Funktion oder Methode kann sowohl textuell in der präprozessierten Datei auf Änderungen geprüft als auch zusätzlich auf Änderungen im Funktions-Interface geprüft werden. Ein Beispiel ist in Bild 5 dargestellt, dort ändert sich der Wert des Präprozessor-Makros „PI“ von einer Version der Software zur nächsten Version. Dadurch ändert sich der präprozessierte Code der Funktion, obwohl sich der Original-Code der Funktion nicht geändert hat. Die Tests für diese Funktion müssen also eindeutig erneut durchgeführt werden.

Wiederholen von Tests

Für die Ermittlung, welche Funktionen beim Regressionstest einer neuen Software-Version erneut getestet werden müssen, sind folgende Prüfungen notwendig:

- Im einfachsten Fall, wenn sich die Funktion im Original-Source-Code geändert hat, müssen die betreffenden Tests wiederholt werden.
- Wenn sich die Funktion im präprozessierten Code geändert hat, ist ebenfalls ein erneuter Test notwendig.
- Etwas komplizierter wird der Fall, wenn sich im präprozessierten Code einer Funktion nichts geändert hat, aber das Interface der Funktion durch Änderungen an verwendeten Elementen modifiziert wurde (z. B. Typänderung einer gelesenen Variable). Auch in diesem Fall ist ein erneuter Test notwendig.

Eine solchermaßen eingeschränkte Liste von erneut durchzuführenden Tests ist in einem Continuous-Integration-Prozess sinnvoll, um die Gesamtlaufzeit der Tests bei häufigen Testläufen möglichst klein zu halten (z. B. zur Verifikation von Code-Änderungen vor dem Commit). Für ein neues Release einer Software wäre trotzdem eine Durchführung aller verfügbaren Tests anzuraten, da man niemals ausschließen kann, dass im Analyse-Prozess ggf. notwendige Regressionstests übersehen wurden.

Fazit

Eine Zertifizierung sicherheitskritischer Software erfordert umfangreiche und normgerechte Tests aller Funktionalitäten. Die Vollständigkeit der Tests wird unter anderem über die Code-Coverage nachgewiesen. Bei Softwarevarianten auch für alle Code-Varianten. Mit einer erweiterten Analyse der Quelldateien lassen sich wertvolle Informationen gewinnen, um nach kleineren Code-Änderungen insgesamt weniger Tests erneut durchführen zu müssen. Auch die Erkennung von identischen Code-Anteilen in Varianten ermöglicht ein Zusammenfassen von Coverage-Ergebnissen aus verschiedenen Tests und damit eine schnellere Erreichung der gewünschten Code-Abdeckung. Die mehrfache Nutzung von Tests in Varianten mit jeweils erweiterten oder angepassten Testdaten verringert ebenfalls den Aufwand für die Testerstellung.

Vollständige Sicht

Die Ermittlung einer Hyper-Coverage aus den verfügbaren Coverage-Ergebnissen erlaubt eine vollständige Sicht der Testabdeckung auf Quelldateiebene, ohne die spezifischen Coverage-Anforderungen der Standards und Normen für die Entwicklung sicherheitskritischer Software zu verwässern. Im Gegenteil: Mit der Hyper-Coverage lassen sich zuverlässig nicht getestete Stellen in den originalen C/C++-Quelldateien erkennen.

Wer schreibt

Michael Wittner ist Geschäftsführer der Razorcat Development GmbH. Der Diplom-Informatiker ist seit mehr als 25 Jahren im Bereich Software-Entwicklung und Test tätig. Nach dem Studium der Informatik an der TU Berlin arbeitete er als wissenschaftlicher Mitarbeiter der Daimler AG an der Entwicklung von Testmethoden und Testwerkzeugen. Seit 1997 ist er geschäftsführender Gesellschafter der Razorcat Development GmbH, dem Hersteller des Unit-Testtools TESSY und CTE sowie des Testmanagement-Tools ITE und der Testspezifikationssprache CCDL. ◄

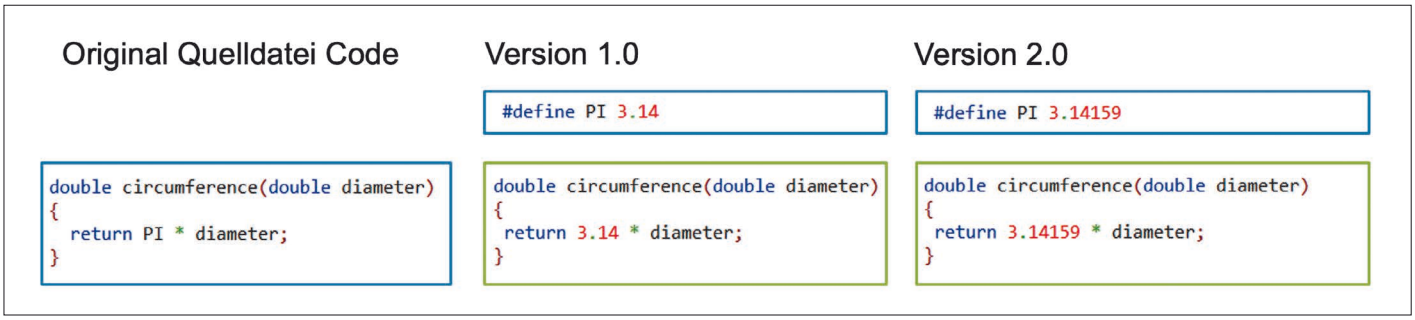

Bild 5: Geänderter Wert eines Präprozessor-Makros