

# Software-Qualität: Statische Code-Analyse und dynamische Tests - Komplementäre Methoden zur Qualitätssicherung

Von Klaus Lambertz (Verifysoft Technology GmbH)

**Um eine gute Softwarequalität zu gewährleisten, werden während der Softwareentwicklung zwei sich ergänzende Verfahren eingesetzt: Statische Code-Analyse und dynamische Tests in Verbindung mit Code-Coverage-Messungen. Dieses Papier zeigt die Vorteile und Grenzen beider Methoden auf und erklärt, warum statische und dynamische Tests einander ergänzen und beide notwendig sind, um eine hohe Softwarequalität zu gewährleisten.**

Die statische Analyse ist eine Überprüfung der internen Struktur einer Software. Es handelt sich hierbei also nicht um eine Bewertung der Funktionalität einer Software. Bei der statischen Analyse wird auf den Code analysiert, ohne dass er ausgeführt wird. Eine solche Analyse kann prinzipiell auch manuell im Rahmen von Walkthroughs oder Inspektionen durchgeführt werden. Die Qualität solcher manuellen Bewertungen ist zwar nützlich, hängt aber von den Mitarbeitern und ihrer Wachsamkeit zum Zeitpunkt der Inspektion ab.

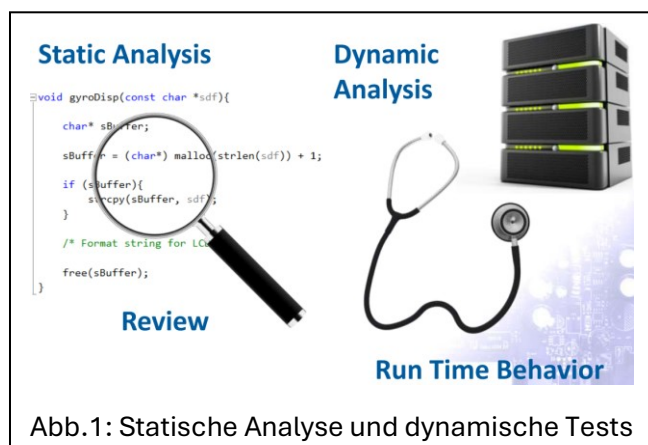


Abb.1: Statische Analyse und dynamische Tests

Aus diesem Grund sollten Walkthroughs und Inspektionen durch automatische Überprüfungen mit Tools zur statischen Codeanalyse ergänzt werden.

Mit „Advanced Static Analysis Tools“ ist es möglich, modulübergreifende Analysen und Analysen für große Codebasen mit Millionen von Codezeilen durchzuführen.

Grob gesagt geht es bei der statischen Analyse um die Überprüfung auf Fehler und die Einhaltung

von Programmierrichtlinien. Hierfür sind keine Testfälle erforderlich - die Analyse wird automatisch vom Tool durchgeführt.

Dynamische Tests hingegen verfolgen den umgekehrten Ansatz. Sie werden ausgeführt, während das Programm in Betrieb ist. Bei diesen Tests wird während der Laufzeit geprüft, ob die Anwendung wie erwartet funktioniert. Anhand von Testfällen wird festgestellt, ob die Software entsprechend den Anforderungen funktioniert. Das heißt, es wird festgestellt, ob die Software das tut, was sie tun soll.

Was die Kosten für Softwaretests angeht, ist es interessant, sich die Untersuchungen von Barry Boehm anzusehen. Obwohl die genauen Zahlen je nach Produkt und Projekt variieren, steigen die Kosten für die Beseitigung von Fehlern über die Dauer des Softwareentwicklungsprozesses stets exponentiell an. Wenn ein Problem in einer frühen Entwicklungsphase entdeckt wird, kann es zu relativ geringen Kosten behoben werden. Wenn ein Fehler während der Implementierung entdeckt wird, kann relativ kostengünstig sogar vom Entwickler selbst behoben werden.

Im schlimmsten Fall wird ein Problem nach der Auslieferung der Software entdeckt. Dies kann zu teuren Rückrufaktionen, Produktionsausfällen, finanziellen Verlusten und Personenschäden führen.

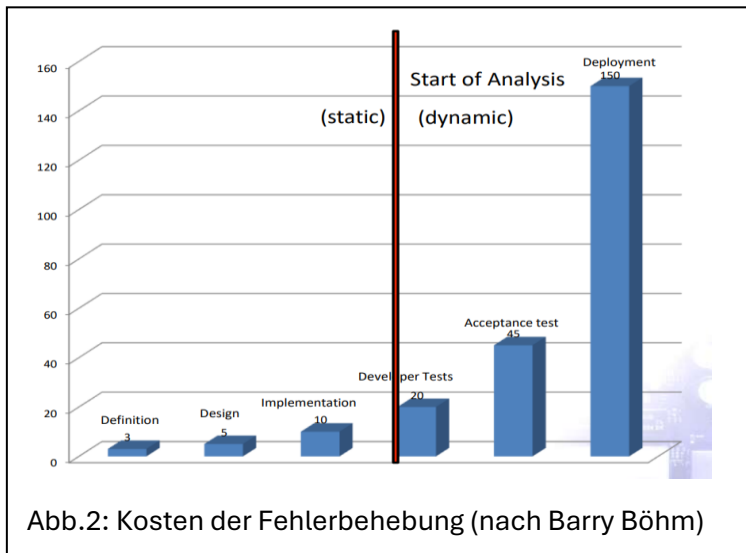


Abb.2: Kosten der Fehlerbehebung (nach Barry Böhm)

Für das Testen bedeutet das: Je früher die Tests durchgeführt werden, desto früher werden Fehler entdeckt. Sie können dann früher behoben werden, wenn die Korrektur noch relativ kostengünstig ist. Da für die dynamische Tests zur Laufzeit die Software ausgeführt werden muss, sind diese Tests erst nach der Implementierung möglich. Die statische Analyse hingegen kann schon während der Implementierung und bereits mit kleinen Codeteilen durchgeführt werden.

Mit der statischen Code-Analyse können Fehler entdeckt werden, bevor dynamische Tests durchgeführt werden. Dies ist unter dem Begriff "shift left" bekannt, einer Praxis, bei der Tests, Qualitäts- und Leistungsbewertung früh im Entwicklungsprozess stattfinden. Sobald ein Tool zur statischen Codeanalyse eingerichtet ist, arbeitet es "von selbst" und ohne Aufwand. Es ist ein guter Ansatz, das Tool jedem Entwickler zur Verfügung zu stellen, so dass der Code erst nach der Analyse eingesehen wird.

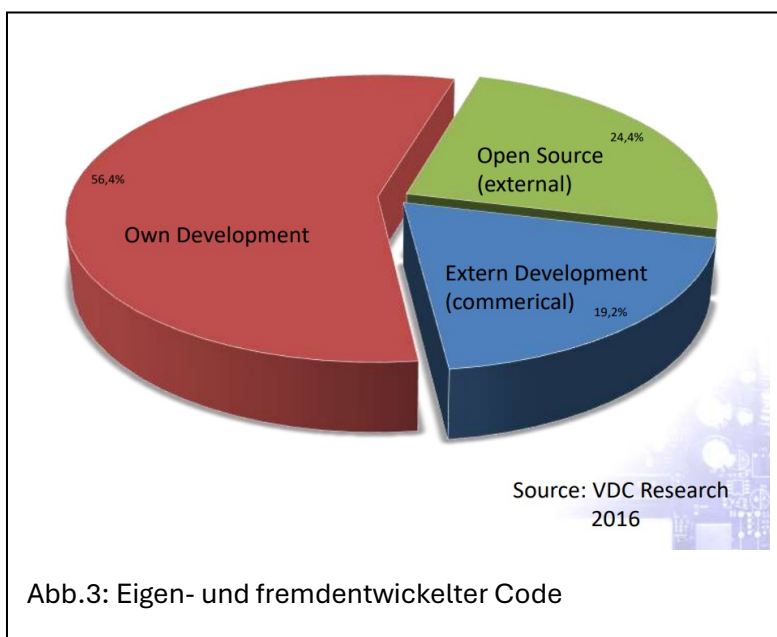


Abb.3: Eigen- und fremdentwickelter Code

Heutzutage wird nur noch ein Teil der Software innerhalb des Unternehmens geschrieben. Laut einer Studie von VDC Research aus dem Jahr 2016 werden nur 56 % des Codes intern implementiert. Der Rest ist Open-Source-Software und extern entwickelte Software von Drittanbietern, für die in der Regel nur die Binärdateien und nicht der Quellcode verfügbar sind. In diesem Zusammenhang ist es gut zu wissen, dass fortschrittliche Werkzeuge für die

statische Analyse nicht nur den Quellcode, sondern auch den Binärcode analysieren können (auch wenn die Qualität der Binäranalyse nicht immer der der Quellcodeanalyse entspricht).

Heutzutage bieten statische Analysetools unter anderem folgende Überprüfungen:

- **Syntax**  
Dies geschieht in einer weitaus besseren Qualität und Tiefe als bei den Syntaxprüfungen von Compilern.
- **Semantik**  
Ein Beispiel: Der Satz "Nachts ist es kälter als draußen" ist syntaktisch korrekt, aber die Semantik ist es nicht, denn dieser Satz ist unsinnig. Die statische Codeanalyse prüft die Semantik jedoch nur in begrenztem Umfang, da das Werkzeug verstehen muss, was im Code gemeint ist.
- **Kontrollflussprobleme**  
z.B. nicht erreichbarer Code, Sprünge in oder aus Schleifen.
- **Datenflussprobleme**  
z.B. nicht initialisierte Variablen.
- **Nebenläufigkeitsprobleme**  
z. B. Race Conditions (wie Data Races), vergessene oder falsch durchgeführte Synchronisierungen und Deadlocks. Nebenläufigkeitsprobleme sind eine Schlüsseldisziplin der statischen Analyse - oft können solche Probleme nur über die statische Analyse und nicht mit der dynamischen Analyse zur Laufzeit nicht erkannt werden.
- **Programmierregeln und Kodierungsstandards**  
Die bekanntesten sind die MISRA-Regeln, aber es gibt noch viele weitere. Viele Sicherheitsstandards verlangen diese Überprüfungen. Gute Tools bieten auch die Möglichkeit, eigene Regeln zu implementieren.
- **Wartbarkeit / Metriken**  
z.B. Halstead und McCabe Metriken. Darüber hinaus gibt es noch viele andere Metriken, sogar solche wie "Estimated Number of Bugs" und "Time to understand a code". Auch wenn die Software nicht zwangsläufig schlecht sein muss wenn bestimmte metrische Schwellenwerte überschritten werden, ist es eine Tatsache, dass es einen Zusammenhang zwischen Codekomplexität und Fehleranfälligkeit, Testbarkeit und Wartbarkeit gibt.
- **Architektur**  
Wie ist die Struktur der Software? Dies ist vor allem bei Legacy Code von Interesse, mit dem niemand mehr vertraut ist.
- **Sicherheit**  
Aufgrund der Konnektivität moderner Software wird die Prüfung auf Sicherheitslücken immer wichtiger. Mit der statischen Codeanalyse wird festgestellt, ob der Code Strukturen enthält, die Angriffe von außen ermöglichen.

## Statische Code-Analyse ist für sicherheitskritische Software obligatorisch

Normen für die Entwicklung sicherheitskritischer Software wie ISO 26262 im Automobilsektor erfordern eine statische Codeanalyse. Kontrollflussanalyse und Datenflussanalyse werden für ASIL C und höher "dringend empfohlen". Die statische Analyse ist für ASIL B und höher obligatorisch und für die niedrige Stufe ASIL A zumindest empfohlen. Die Empfehlungen anderer Normen sind ähnlich.

Methoden	ASIL			
	A	B	C	D
Walk-through	++	+	o	o
Inspection	+	++	++	++
Semi-formal verification	+	+	++	++
Formal verification	o	o	+	+
Control flow analysis	+	+	++	++
Data flow analysis	+	+	++	++
Static code analysis	+	++	++	++
Semantic code analysis	+	+	+	+

+ recommended    ++ highly recommended

Abb.4: Statische Analyseverfahren nach ISO 26262

## Statische Code-Analyse-Tools bieten viele Checks

Für die funktionale Sicherheit haben statische Code-Analyse-Tools viele Prüfer, darunter für Speicherüberläufe, Null-Zeiger-Dereferenzierung, Division durch Null, Speicherfreigabe von Nicht-Heap-Variablen, Speicherlecks, fehlende Return-Anweisungen und vieles mehr.

Zu den Sicherheitsprüfungen gehören Pufferüberläufe, unterminierte C-Strings, Additions-, Multiplikations- und Subtraktionsüberläufe der Allokationsgröße, hartkodierte DNS-Namen, Klartextspeicherung von Passwörtern, Verwendung von `system()` und vieles mehr.

Zusammengefasst sind dies die Hauptvorteile der statischen Code-Analyse:

- kann in einem frühen Stadium des Entwicklungsprozesses durchgeführt werden
- bietet eine Vielzahl verschiedener Checker
- die Analyse erfolgt automatisch und in der Regel ohne jeglichen Aufwand

## Code Coverage ist obligatorisch, um sicherzustellen, dass der gesamte Code getestet wird.

Ergänzend zur statischen Code-Analyse werden Tests zur Laufzeit durchgeführt: die dynamischen Tests.

Beim dynamischen Testen wird die Anwendung mit Testfällen gefüttert, die auf der Ebene der Einheitstests beginnen. Die Ergebnisse der Tests werden mit den erwarteten Ergebnissen verglichen, und so wird geprüft, ob die Anwendung so funktioniert, wie sie funktionieren soll. Hierfür ist die Qualität der Tests wichtig. Die Tests sollten sich an den Anforderungen orientieren.

Insbesondere Grenzwerte und Randbedingungen sollten getestet werden. Es ist auch wichtig, die Codeabdeckung zu messen, um zu sehen, ob die gesamte Anwendung getestet wurde.

Code Coverage zeigt, ob die Software vollständig getestet wurde. Bleiben Teile des Codes ungetestet, können diese Codeteile gefährliche Fehler enthalten, die in der Anwendung versteckt sind - und das sollte vermieden werden.

Allerdings gibt die Codeabdeckung keine Auskunft über die Qualität der Tests. Einfach "irgendwelche" Tests durchzuführen, um den Abdeckungsgrad zu erhöhen, ist kein guter Ansatz.

Die Frage ist immer, was "vollständige Tests" oder "100%ige Abdeckung" bedeutet. Dies hängt vom gewählten oder erforderlichen Abdeckungsgrad (der Code Coverage Stufe) ab.

Als Ausgangspunkt kann man davon ausgehen, dass jede Anweisung einmal getestet wurde. Dies ist die Statement Coverage - ein relativ schwacher Abdeckungsgrad.

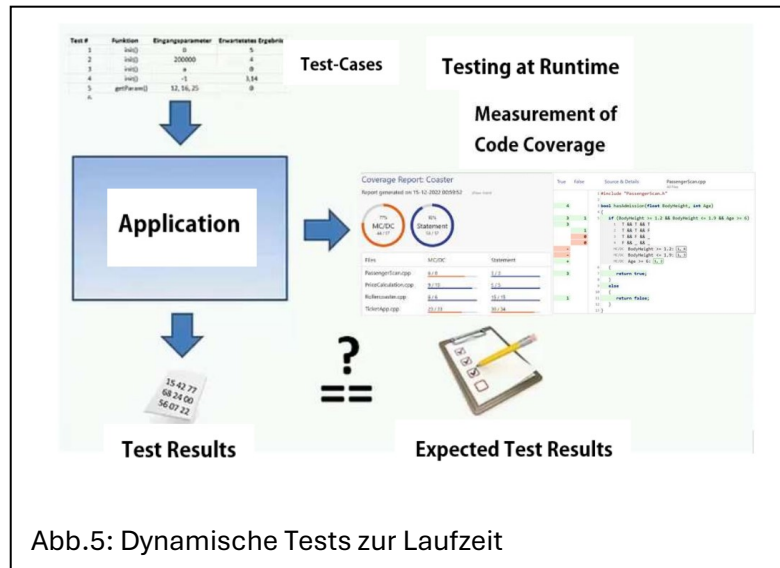


Abb.5: Dynamische Tests zur Laufzeit

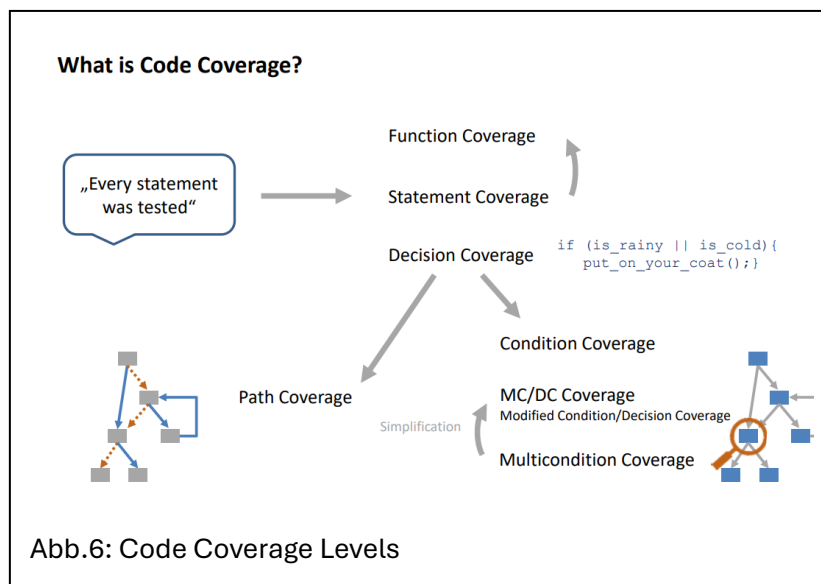


Abb.6: Code Coverage Levels

Function Coverage ist noch schwächer.

Decision Coverage und Condition Coverage sind schon stärker, und Multicondition Coverage ist sehr streng. Hier müssen alle möglichen Kombination aller atomaren Bedingungen als "wahr" und als "falsch" bewertet werden. Bei einer Software mit vielen atomaren Bedingungen führt dies zu einer hohen Anzahl von Testfällen. Aus diesem Grund gibt

es eine Vereinfachung der Multicondition Coverage: die Modified Condition/Decision Coverage (MC/DC Coverage). Bei dieser Testabdeckungsstufen werden mit weniger Testfällen die gleichen Testergebnisse geliefert. Die MC/DC Coverage ist die höchste von den Sicherheitsnormen geforderte Stufe.

Allen Sicherheitsnormen ist gemeinsam: je höher die Sicherheitsstufe, desto höher der erforderliche Abdeckungsgrad. Die ISO 26262 fordert für ASIL D, dem höchsten Automotive Safety Level, MC/DC-Coverage.

Das Gleiche gilt für die Normen für die anderen Sektoren, z. B. in der Luftfahrt- und Eisenbahnindustrie, sowie für die "allgemeine" Norm IEC 61508. Die höchste geforderte Stufe ist immer MC/DC-Deckung.

**Code Coverage Requirements of Standards like ISO 26262**

Coverage Level	ASIL A	ASIL B	ASIL C	ASIL D
Statement	++	++	+	+
Branch	+	++	++	++
MC/DC	+	+	+	++

Similar for standards in other industries

↑

Multicondition


MC/DC

Condition


Decision

Statement


Function



DO-178C



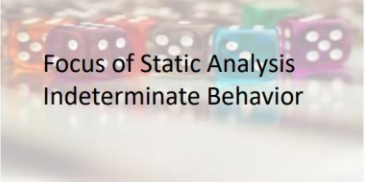
EN 50128



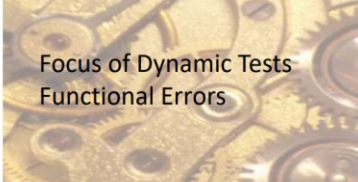
IEC 61508

Abb.7: Code Coverage und Sicherheitsstandards


## Statische Code-Analyse und dynamische Tests verbunden mit der Messung der Code-Abdeckung



Focus of Static Analysis  
Indeterminate Behavior



Focus of Dynamic Tests  
Functional Errors



Particularly  
Concurrency  
Problems

Static Code Analysis and dynamic tests are **complementary**.

The **use of both methods** is absolutely necessary to develop good software!

Abb.8: Statische Analyse und dynamische Tests

Zusammenfassend kann man sagen: Hauptanwendungsgebiet der statischen Codeanalyse ist die Erkennung von Fehlern, die zu undefiniertem Verhalten führen. Eine besondere Stärke ist das Aufdecken von Nebenläufigkeitsproblemen. Wenn wir von statischer Codeanalyse sprechen, geht es in erster Linie um nichtdeterministische Fehler. Mit anderen Worten: Fehler, die nicht immer auftreten. Das ist der Grund, warum sie

mit Tests während der Laufzeit nur schwer zu erkennen sind. Manchmal treten diese nicht-deterministischen Fehler erst nach Jahren auf, manchmal nie, aber sie können jederzeit auftreten. Die statische Codeanalyse deckt zwar schwerwiegende Probleme im Quellcode auf, kann aber keine Auskunft über die korrekte Funktionalität der Software geben. Dies kann durch die dynamischen Tests erfolgen, sobald ein Programm (oder ein Teil davon) ausgeführt werden kann. Statische und dynamische Analyse sind beide unerlässlich und müssen komplementär eingesetzt werden. Es geht nicht um entweder oder - es geht um beides!

## Auswahlkriterien für die Werkzeuge

Im Folgenden werde ich einige Auswahlkriterien für die Auswahl von Tools beschreiben. Zunächst wollen wir uns, wie im Entwicklungsprozess, die Werkzeuge für die statische Codeanalyse ansehen. Wir haben gesehen, dass diese Werkzeuge große Vorteile haben: Einsatz zu einem frühen Zeitpunkt im Entwicklungsprozess, Überprüfung auf Programmierrichtlinien und Erkennung von Fehlern ohne das Schreiben von Testfällen.

Es gibt jedoch eine Sache, die das Aufdecken von Fehlern erschwert: die statische Code-Analyse erkennt Fehler nicht immer 100%ig genau.

Wenn das Tool "True Positives" und "True Negatives" meldet, ist alles in Ordnung. True Positives sind Fehler, die vom Tool angezeigt werden und die „echte“ im Code vorhandene Fehler sind, die behoben werden sollten. True Negatives bedeutet, dass das Tool keinen Fehler meldet und auch kein Fehler im Code vorhanden ist - auch das ist in Ordnung.

Das Diagramm zeigt vier Kategorien von Fehlermeldungen:

- True Positive**: A "real" bug was detected. 🟢
- False Positive**: A bug was reported, but there is no bug. 🟡
- False Negative**: A "real" bug was not detected. 🚫🚫
- True Negative**: There is no bug, and no bug has been reported. No false alarm. 🟢

Ein rotes Textfeld oben rechts enthält den Text: "A bug which should be fixed!"

Abb.9: Qualifizierung von Fehlermeldungen

Schwieriger wird es bei "False Negatives" und "False Positives". False Negatives sind tatsächlich vorhandene Fehler, die vom Tool nicht gefunden werden. False Positives sind "Fehler", die das Tool anzeigt, die aber keine sind. Es handelt sich also um falsche Warnungen bzw. „Fehlalarme“. Sie werden natürlich die "True Positives" beheben, aber Sie werden sich auch die "False Positives" ansehen müssen, da Sie nicht wissen, dass es sich um „Fehlalarme“ handelt. Das bedeutet, dass Sie alle Fehlermeldungen überprüfen müssen, unabhängig davon, ob es sich um tatsächliche Fehler oder um False Positives handelt. Das Problem bei den False Positives ist, dass sie Ihre Zeit in Anspruch nehmen. Zudem besteht die Gefahr, dass Sie nach fünf False Positives nicht mehr auf den sechsten achten, bei dem es sich um einen echten Fehler handeln könnte... Daher ist es wichtig, ein Tool für die statische Codeanalyse zu haben, das so viele echte Fehler wie möglich findet (True Positives), das keine (oder nur sehr wenige) echte Fehler übersieht (False Negatives) und das keine (oder nur sehr wenige) False Positives zeigt.

Bei der Auswahl eines Werkzeugs ist es auch wichtig zu berücksichtigen, welche Prüfungen, Programmierregeln und Metriken unterstützt werden. Die Möglichkeit, eigene Regeln zu implementieren, ist ein Vorteil. Sinnvoll ist, dass bestimmte Prüfungen und Regeln deaktiviert werden können, um nicht gleich beim ersten Durchlauf des Tools zu viele Meldungen zu erhalten. So hat der Tester die Möglichkeit, sich zunächst auf die wichtigsten Fehler zu konzentrieren. Aus dem gleichen Grund ist es gut, wenn Teile des Projekts bei der Analyse ausgeschlossen werden können.

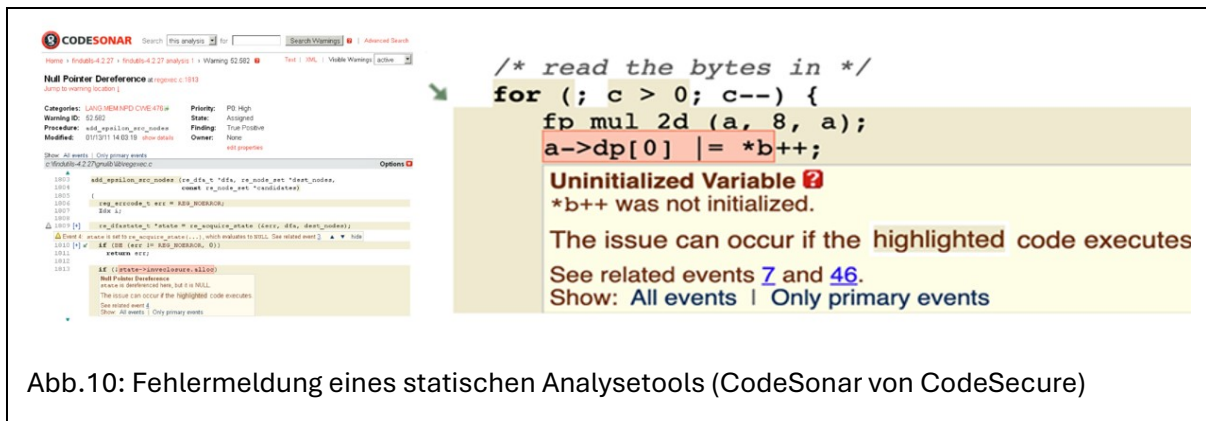


Abb.10: Fehlermeldung eines statischen Analysetools (CodeSonar von CodeSecure)

Eine gute Darstellung der Fehlermeldungen ist wichtig für das Verständnis. So hilft sie, Zeit zu sparen. Es sollte die Möglichkeit bestehen, die Meldungen zu klassifizieren und es sollte die notwendige Hilfe geben, um die Warnungen zu qualifizieren. Es ist wichtig, dass die Meldungen einzelnen Teammitgliedern zur Behebung zugewiesen werden können. Außerdem sollte es Regeln für den Zugriff auf die Ergebnisse geben.

Insbesondere bei größeren Codebasen ist die Performance des Werkzeugs für die statische Codeanalyse wichtig. Bei der Auswahl eines Tools sollte geprüft werden, ob größere Codebasen mit mehreren Millionen Codezeilen in einem angemessenen Zeitrahmen analysiert werden können. Um dieses Ziel zu erreichen, ist es hilfreich, wenn die Analyse auf mehrere Rechner verteilt werden kann. Eine inkrementelle Analyse sollte möglich sein. Die benötigte Hardware und der erforderliche Speicherplatz sollten evaluiert werden.

Bei der Auswahl eines Code Coverage Analyzers muss selbstverständlich darauf geachtet werden, dass das Tool die geforderten Code Coverage Stufen unterstützt. So ist für ein ASIL D-Projekt in der Automobilindustrie eine MC/DC-Abdeckung zwingend erforderlich. Auch wenn Sie im aktuellen Projekt ein niedrigeres Sicherheitsniveau haben, sollten Sie ein Werkzeug in Betracht ziehen, das auch höhere Abdeckungsgrade analysieren kann, da Sie später möglicherweise Projekte mit strengeren Anforderungen haben. Durch vorausschauende Entscheidungen bei der Anschaffung eines Werkzeugs wird vermieden, dass Tools zweimal gekauft werden müssen: zuerst ein billiges und später ein gutes.

Der Code-Coverage-Analyzer muss natürlich mit dem von Ihnen verwendeten Compiler funktionieren. Aber auch hier ist es - aus den oben genannten Gründen - sinnvoll, ein Tool zu wählen, das mit mehreren Compilern oder besser mit allen Compilern zusammenarbeitet.

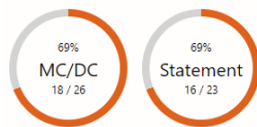
Das Werkzeug sollte in der Lage sein, mit Ihren embedded Targets zu arbeiten. Aus diesem Grund ist es wichtig, dass das Tool einen geringen Instrumentierungs-Overhead hat. Die Instrumentierung ist das Hinzufügen von Zählern zu Ihrem Code (oder einer Kopie Ihres Codes), um die Abdeckung zu messen.

Wie bei der statischen Codeanalyse ist es auch für das Code Coverage Tool wichtig, wie die Berichte aussehen. Sie sollten klar und einfach zu verstehen sein und sowohl einen Überblick über die Code Coverage über das gesamte Projekt als auch die Details der Abdeckung in Ihrem Quellcode zeigen.

## Comprehensible and meaningful reports

### Coverage Report: My Home Control AI

Report generated on 30-11-2022 16:24:38 [show more](#)



Files and Functions	MC/DC	Statement
lights()	6 / 8	5 / 6
lights()	6 / 8	5 / 6
close_windows()	2 / 2	1 / 1
open_windows()	2 / 2	1 / 1
open_windows_for()	2 / 2	3 / 3
heat()	2 / 2	1 / 1
air_condition()	0 / 2	0 / 1
temperature_control()	4 / 8	5 / 10

True	False	Source & Details	regulators.c All Files
3		1 #include "regulators.h"	
		2 #include "sensors.h"	
		3	
		4 void lights(enum light_status goal)	
		5 {	
	3	6 if (goal == off)	
		7 {	
		8 printf("Light is switched off.\n");	
		9 }	
2	1	10 else if (goal == on)	
		11 {	
		12 printf("Light is switched on.\n");	
		13 }	
		14 else if (goal == dimmed)	
		15 {	
		16 printf("Lights are romantically dimmed.\n");	
		17 }	
		18 }	

Abb.11: Code Coverage Report (Testwell CTC++ von Verifysoft Technology)

Manchmal ist es nicht möglich, alle Codeteile mit Tests zu erreichen - zum Beispiel, wenn Sie einen defensiven Programmierstil anwenden müssen. In diesem Fall ist es hilfreich, wenn das Tool die Möglichkeit bietet, "Justifications" hinzuzufügen, um zu erklären, warum bestimmte Teile des Codes nicht durch Tests abgedeckt wurden.

Wenn Sie ein Code Coverage Tool für sicherheitskritische Projekte verwenden, müssen Sie nachweisen, dass das verwendete Tool qualifiziert ist. Wenn das Werkzeug über ein Zertifikat verfügt (z. B. ein TÜV-Zertifikat), kann es für zahlreiche Sicherheitsstandards ohne weitere Qualifizierungsmaßnahmen verwendet werden. Bei DO178-C-Projekten in der Luftfahrt ist die Situation etwas anders. Das Werkzeug muss auf jeden Fall qualifiziert werden, aber auch hier ist das Werkzeugzertifikat als Ausgangspunkt hilfreich – der Toolhersteller bietet in der Regel zusätzlich Unterstützung und weitere Testfälle an.

Diese Qualifikation/Zertifizierung ist auch für Werkzeuge zur statischen Codeanalyse erforderlich.

Die folgenden Auswahlkriterien betreffen sowohl statische als auch dynamische Analysewerkzeuge:

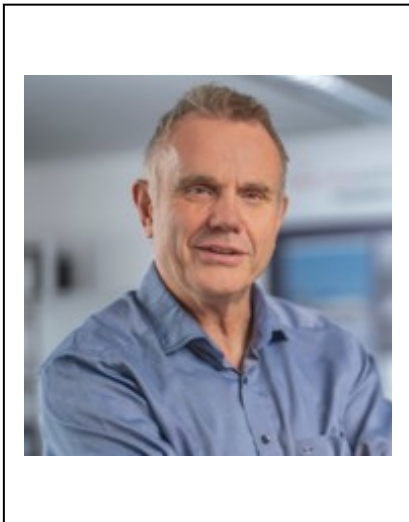
Wichtig sind immer die Kosten und das Lizenzierungsmodell für eine Test- und Analysesoftware. Wenn wir über Kosten sprechen, geht es nicht nur um den Preis des Tools. Es geht auch um die Kosten für die Einrichtung des Tools, die Schulung der Benutzer und die Zeit, die Sie benötigen, um die Berichte zu verstehen. Hier kann ein "billiges" Tool teuer werden... Ein "billiges" Tool, das eine Menge Probleme verursacht, ist letztendlich nicht billig... Es ist auch wichtig zu prüfen, wie groß der Einsatzbereich eines Tools ist: ein Arbeitsplatz, das gesamte Projekt, ein Standort oder sogar eine weltweite Nutzung innerhalb Ihrer Organisation?

Die Integration in CI-Prozesse ist ebenfalls wichtig.

Bei der Auswahl eines Tools sollten Sie auch auf die Qualität des Supports achten. Bei einer Tool-Evaluierung können Sie sich bereits ein Bild von der Reaktionszeit und der Zusammenarbeit zwischen Anwender und Tool-Anbieter machen.

Zusammenfassend haben wir gesehen, dass statische Code-Analyse und dynamische Tests im Zusammenhang mit der Code-Abdeckung notwendig sind und komplementär eingesetzt werden müssen. Es gibt sehr gute Tools auf dem Markt, die Sie bei diesen Methoden unterstützen. Vor dem Erwerb von Lizenzen sollten die Tools evaluiert werden, um zu prüfen, ob sie den Anforderungen Ihres aktuellen Projekts und möglicher zukünftiger Projekte entsprechen.

## Über den Autor



Klaus Lambertz ist Gründer und Geschäftsführer der Verifysoft Technology GmbH [www.verifysoft.com](http://www.verifysoft.com), einem Unternehmen, das Softwaretest- und Analysewerkzeuge anbietet. Verifysoft Technology wurde 2003 in Offenburg (Deutschland) gegründet und hat derzeit mehr als 750 zufriedene Kunden in über 40 Ländern. Neben dem Code Coverage Analyzer Testwell CTC++ bietet das Unternehmen ergänzende Werkzeuge für die statische Codeanalyse an. Verifysoft bietet auch Seminare zu Themen des Softwaretestens an.

Teile dieses Artikels basieren auf Veröffentlichungen von Royd Lüdtke (Director Static Code Analysis, Verifysoft) und Dr. Sabine Pöhler (Product Manager Testwell CTC++, Verifysoft).