Statische Codeanalyse

Fabian Feder
Hochschule Offenburg
Seminar Neue Technologien
Badstraße 24, 77652, Offenburg

01.08.2012

***Zusammenfassung*— Automatisierte Codeüberprüfung ist mittlerweile Stand der Technik. Sie wird in vielen Bereichen der Softwareentwicklung eingesetzt, von Embedded Anwendungen bis hin zu großer Unternehmenssoftware. Im sicherheitskritischen Bereich ist es jedoch notwendig zu beweisen, dass die Tests korrekte Ergebnisse liefern. Dies ist mit der „statischen Codeanalyse" möglich. Es gibt entsprechende Software, welche mathematisch beweist, dass bestimmte Fehler nicht vorhanden sind.**

I. Einleitung

Fehlerhafte Programme, unentdeckte Fehler in Programmen gibt es nicht erst seit Windows. Es gab schon zuvor Fehler und es wird durch die steigende Komplexität der Software immer Fehler geben. Jedoch ist die Softwareentwicklung mittlerweile soweit, dass es einfache, komfortable und automatisierte Testsysteme gibt. Sie sind für den Entwickler ein gutes Werkzeug um den Großteil der Fehler zu finden. Hier kann man auf das Pareto-Prinzip[1] verweisen. Mit 20% Aufwand wird man ca. 80% der Fehler finden. Um jedoch hundertprozentige Fehlerfreiheit zu garantieren (wenn überhaupt möglich), muss man sehr großen Aufwand betreiben.

Auf erster Ebene der automatischen Fehlererkennung befindet sich der Compiler und Linker, welche statisch Syntaxfehler und Coderichtlinien kontrollieren. Dies reicht jedoch nicht um fehlerfreien Code zu gewährleisten. Kent Beck beschrieb 1998 in seinem Buch die Grundlagen der xUnit Familie [Bec98], welche einen weiteren Kontrollschritt darstellt.

Seit dem gibt es eine Vielzahl von Unit-Test-Frameworks, wie zum Beispiel JUnit für Java und NUnit für die .net-Umgebung. Durch sie wird es ermöglicht individuelle, automatisierte und dynamische Tests durchzuführen. Diese Tests sind jedoch wiederum nur so gut, wie sie der Programmierer geschrieben hat. Falls er nicht alle Fehlermöglichkeiten bedacht hat, kann die Testumgebung nicht alle Fehler finden und suggeriert so, dem unerfahrenen Entwickler, eine scheinbare Sicherheit vor. Code-Coverage unterstützt hierbei den Entwickler mit einer farbigen Überdeckung des Quellcodes um festzustellen ob alle relevanten Codestücke von Testfällen ausgeführt werden. Wie schon zuvor erhält man auch hierdurch keine hundertprozentige Sicherheit. An diesem Beispiel von Büchner

```
1  int i;
2  for(i = 0; i < 2; i++){
3    switch(i){
4      case 0:
5        a = 600;
6        break;
7      case 1:
8        a = 700;
9        break;
10   }
11 }
```

erkennt man, dass der ganze Code durchlaufen wird, jedoch wird die 100%ige Zweigabdeckung nicht erreicht. Das liegt am fehlenden Durchlauf des „**default**:"-case in der Switch-Anweisung. Dieser „**default**:"-Zweig existiert immer, egal ob er explizit programmiert wurde oder nicht [Bue10b], [Bue10a].

Durch die Ausführung des Quellcodes bei der dynamischen Analyse sind Tests möglich, die nur im laufenden Betrieb zu messen sind. Anhand dieser, kann man die Laufzeit oder den Speicherverbrauch eines Programms bestimmen. Dadurch kann man Rückschlüsse ziehen, an welcher Stelle im Programm eventuell Optimierungen durchzuführen oder Fehler zu finden sind.

II. Statische Codeanalyse

Die statische Analyse findet bereits im Compiler statt. Es werden sehr einfache Dinge wie auch kompliziertere Fehlerquellen überprüft. Zum Beispiel:

- Wurde die Variable bereits vor Verwendung initialisiert?

[1] Vilfredo Pareto untersuchte im 19. Jahrhundert die Verteilung des Volksvermögens in Italien und fand heraus, dass ca. 20 % der Familien ca. 80 % des Vermögens verfügten.

- Sind Variablen doppelt deklariert?
- Wurde der Programmierstil eingehalten?

Wie die drei einfachen Fehlertypen zeigen, wäre es einfach diese manuell zu prüfen. Es ist jedoch bequemer, fehlerfreier und zeitsparender, diese monotone Arbeit automatisiert überprüfen zu lassen.

Ein etwas komplexeres Beispiel von Schornböck [Sch04] zeigt, dass bereits Compiler falschen Code evtl. erkennen können, vorausgesetzt man hat die richtigen Schalter des Compilers aktiviert. In diesem Fall: “gcc -W -Wall -pedantic -std=c++98”

```
1  #include <iostream>
2  using namespace std;
3  int zahl()
4  {
5      return 3;
6  }
7  int main()
8  {
9      cout<<zahl;
10 }
```

Als Warnung des gcc Compilers erhält man für Zeile 9: „warning: the address of ’int zahl()’, will always evaluate as ’true’ ”. Somit erhält man statt der 3 eine 1. Der Programmcode an sich funktioniert fehlerfrei, lediglich der Funktionsaufruf findet nicht statt. Wäre der Rückgabewert eine 1, würde dies dem Programmierer gar nicht auffallen. Durch die Warnung erhält er einen Hinweis auf ein mögliches fehlerhaftes Verhalten des Programms. Bei dieser Art von Analyse sollte ein funktionsfähiges Programm vorliegen, um nicht durch zu viele unnötige Warnungen und Fehler die Übersicht zu verlieren.

Bedingt durch die statische Analyse muss kein lauffähiges Programm vorliegen, da der Quellcode nicht ausgeführt wird. Dies hat den großen Vorteil, dass man bereits unvollständige Programmzeilen testen kann. Da keine selbstgeschriebenen Tests oder Spezifikationen notwendig sind, kann dies als weitere Fehlerquelle ausgeschlossen werden.

Mittels statischer Analyse lassen sich Aussagen über bestimmte Eigenschaften wie Datenfluss, Kontrollfluss und Echtzeit verhalten treffen [SSC04]. Hierbei wird ein fehlerhaftes Programm niemals als korrekt deklariert. Im umgekehrten Fall kann es aber passieren, dass ein richtiges Programm möglicherweise als reparaturbedürftig erkannt wird.

Wie auch bei der dynamischen, gibt es bei der statischen Analyse verschiedene Untersuchungsmöglichkeiten. Abbildung 1 zeigt wie Liggesmeyer die Techniken unterteilt [Lig09].

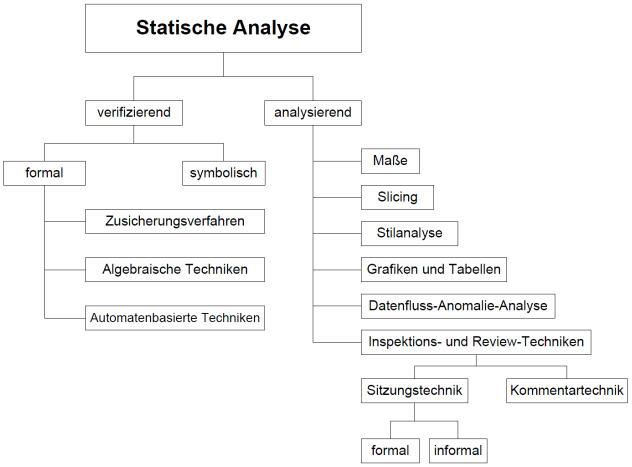

Abbildung 1 – Statische Codeanalyse

Zu den wichtigsten Techniken der statischen Analyse nun ein paar Erläuterungen.

A. *Syntax- / Semantik-Analyse*

Um den Quellcode anhand von bestimmten Richtlinien zu überprüfen, gibt es zwei Gruppen von Methoden.

Syntax-Analyse: Der Quelltext wird anhand von Grammatik- und Syntaxregeln auf seiner Korrektheit überprüft. Dies geschieht in der Regel vom Compiler, der für die jeweilige Sprache angepasst ist.

Semantik-Analyse: Diese Verifikation betrachtet die inhaltliche Bedeutung des Programmcodes [Pli09]. Es wird das Ziel verfolgt, fehleranfällige und generell fragwürdige Sprachkonstrukte mit Hilfe statischer Analysetechniken zu identifizieren [Hof08]. Hierzu ein kleines Beispiel.

```
1  if(1){
2      // Wird immer ausgeführt
3  }else{
4      // Wird nie ausgeführt
5  }
```

Der „**else**”-Zweig, wird nie ausgeführt, da die erste Bedingung immer wahr ist. Somit wird dieser Zweig als Fehler gekennzeichnet.

B. *Slicing*

Bei Slicing wird das Programm in Scheiben zerlegt, bis diese nur noch von einzelnen in sich abhängigen Programmteilen bestehen [Bal09]. Somit kann man sich auf die wesentlichen Programmteile konzentrieren. Hierzu wieder

Beispiel.

```
1 int i;
2 int count = 0;
3 for(i=0; i<=10; i++){
4     count = count + 1;
5 }
6 return count;
```

Angenommen es wird als Rückgabewert eine 10 erwartet. Stattdessen ist der Wert eine 11. Nun ist ein Slice für die Variable “count” durchgeführen, damit die Übersicht verbessert wird:

```
1
2 int count = 0;
3
4     count = count + 1;
5
6 return count;
```

Durch die Reduzierung des Programmcodes kann man nun erkennen, dass es nicht an der Variable “count” liegt. Der Fehler muss also in der Schleife liegen. Für weitergehende Informationen [GGS08].

C. Datenflussanomalien

Datenflussanomalien bezeichnet man als Abfolge von fragwürdigen Variablenzugriffen. Bereits der Compiler weist auf solche Unstimmigkeiten hin. Hoffmann hat die Zugriffe auf eine Variable in drei Kategorien eingeteilt [Hof08]:

- d: Definition einer Variable
- r: Referenzierung einer Variable
- u: Undefinierter Zustand einer Variable

Durch diese Zustände ergibt sich folgende Tabelle:

Muster	Beschreibung	Anomalie?
dd	**Variable wird zweimal hintereinander überschrieben**	**Ja**
dr	Variable wird überschrieben und anschließend verwendet	nein
du	**Variable wird beschrieben und anschließend gelöscht**	**Ja**
rd	Variable wird zunächst verwendet und dann überschrieben	nein
rr	Variable wird zweimal hintereinander verwendet	nein
ru	Variable wird verwendet, dann gelöscht	nein
ud	Undefinierte Variable wird überschrieben	nein
ur	**Undefinierte Variable wird verwendet**	**Ja**
uu	Undefinierte Variable wird gelöscht	nein

Tabelle I – Datenflussanomalie

D. Maße

Es gibt verschiedene Möglichkeiten den Quelltext qualitativ und quantitativ zu erfassen. Diese müssen selbstverständlich deterministisch sein.

- BLOC (Brutto Lines of Code): Anzahl aller Zeilen
- NLOC (Netto Lines of Code): Anzahl der Zeilen, die keine Kommentarzeilen sind
- Anzahl der Kommentarzeilen
- DLOC (Delta Lines of Code): Anzahl der Zeilen, die in der neuen Version eingefügt oder geändert wurden
- Anzahl der Programmzweige (C1)
- Anzahl der Abfragen (C2)
- Anzahl der Programmpfade (C3)
- Schachtelungstiefe
- Komplexitätszahl nach McCabe

[Wim10]

E. Symbolischer Test

Bei dieser Analyse werden symbolische Werte (z.B. X, X+1, Y mit Definitionsbereich) verwendet um die Tests durchzuführen. Im Gegensatz zu der dynamischen Analyse, bei der konkrete Werte eingesetzt werden. Dadurch besteht die Möglichkeit, falls bestimmte Voraussetzungen erfüllt sind, die Korrektheit des Programms zu beweisen [Lig09].
Ein Problem sind die neuen Programmiersprachen, welche nicht symbolisch getestet werden können, da sie einige Sprachkonstrukte (Schleifen, arithmetischer Überlauf) haben, welche im Beweisverfahren nicht dargestellt werden können [Mei06].

F. Abstrakte Interpretation

Eine weitere statische Methode ist die abstrakte Interpretation. Entwickelt wurde das mathematische Verfahren von Patrick Cousot und Radhia Cousot in den siebziger Jahren. Es ist die Theorie der Approximation mathematischer Strukturen zur Beschreibung der Semantik eines Programms [Rei10]. Da viele dynamische Eigenschaften auf entsprechende Zustandsmengen abgebildet werden können, lässt sich deren Gültigkeit durch eine einfache Mengenoperation überprüfen und aus der Schnittmenge im Fehlerfall ein Gegenbeispiel erzeugen [Hof08]. Die Technik abstrahiert nicht nur mathematisch, sondern interpretiert diese Abstraktion auch. Um das Ergebnis zu erhalten, wird eine bestimmte Rechenleistung benötigt, welche in der Vergangenheit nicht zur Verfügung stand. Durch die Leistungssteigerung der Prozessoren und der Verwendung von nicht exponentiellen Algorithmen lassen sich heutzutage die Analysen schon zur Übersetzungszeit der Programme erhalten [Mat07].

Einige statische Codeanalysetools bedienen sich der abstrakten Interpretationen, um möglichst genaue und schnelle Berechnungen ausführen zu können. Leider stehen diese beiden Eigenschaften im Widerspruch. Genauer bedingt hier langsamer. Bei mehreren Variablen entwickelt sich die Zustandsmenge multiplikativ, dadurch muss man die Komplexität auf das Wesentliche reduzieren.

Gegeben ist die Menge $M = \{(+),(-),(\pm),(0)\}$. In der oberen Tabelle ist die abstrakte Multiplikation der Menge M abgebildet. Die Ergebnisse der Addition finden sich in der unteren Tabelle.

$\odot$	$(+)$	$(-)$	(0)	$(\pm)$
$(+)$	$(+)$	$(-)$	(0)	$(\pm)$
$(-)$	$(-)$	$(+)$	(0)	$(\pm)$
(0)	(0)	(0)	(0)	(0)
$(\pm)$	$(\pm)$	$(\pm)$	(0)	$(\pm)$

$\oplus$	$(+)$	$(-)$	(0)	$(\pm)$
$(+)$	$(+)$	$(\pm)$	$(+)$	$(\pm)$
$(-)$	$(\pm)$	$(-)$	$(-)$	$(\pm)$
(0)	$(+)$	$(-)$	(0)	$(\pm)$
$(\pm)$	$(\pm)$	$(\pm)$	$(\pm)$	$(\pm)$

Tabelle II – abstrakte Multiplikation und Addition

Um das Vorzeichen zu bestimmen wird mit deren Eigenschaften und nicht mit deren Werten gerechnet. Daraus ergibt sich im ersten Beispiel ein eindeutiges Ergebnis.

$-2 \cdot 3 = -6$

$\Longrightarrow -(+) \odot (+) = (-) \odot (+) = (-)$

In der zweiten Rechnung kann schon keine eindeutige Aussage mehr gemacht werden.

$-2 + 3 = 1$

$\Longrightarrow -(+) \oplus (+) = (-) \oplus (+) = (\pm)$

Dieses Problem können auch statischen Codeanalysewerkzeuge, die mit abstrakter Interpretation arbeiten, nicht lösen. Sie können für bestimmte Dinge deren Korrektheit beweisen, aber nicht für alle.

Vor und Nachteile Statische Code-Analyse

Vorteile:

- Analyse von Codefragmenten, Methoden, Klassen oder gesamter Software
- Einfache Anwendung, da keine Tests geschrieben werden müssen
- Prüfungen können automatisiert ausgeführt werden
- Einfache Vergleichbarkeit der Messergebnisse mit anderer Software oder Versionen [SS07]
- Vollständige Analyse des Systems

Nachteile:

- Mögliche Fehlermeldungen bei korrektem Code
- Ersetzt nicht die dynamischen Tests
- Es kann keine Aussage über die Funktionalität der Software abgeleitet werden (Validierung) [SS07]
- Es werden nicht alle Programmiersprachen unterstützt

III. Fazit

Die Theorie, der statischen Codeanalyse, aus den siebziger Jahren wurde mittlerweile in verschiedenen Softwareprodukten umgesetzt. Sie ist ein wichtiger Beitrag um die Softwarequalität weiter zu erhöhen, ersetzen kann sie die anderen Testmethoden bislang leider noch nicht. Sie ist somit als zusätzliche qualitätssichernde Maßnahme zu sehen. Als ein weiterer Punkt in der Testkette wird diese Analyse überwiegend in sicherheitskritischen Bereichen verwendet, da die weiteren Kosten nicht in jedem Softwareprodukt zu rechtfertigen sind.

Frank Listing merkt an, dass die statische Untersuchung Fehler bei der Modularisierung in der Designphase finden, aber nicht in ihrer Tragweite bewerten und schon gar nicht eliminieren kann [Lis08]. Auch die Analyse von dreifach geschachtelten Zeigerstrukturen ist nicht so einfach möglich. Das schränkt die Überprüfung von ASCET[2] generierten Code erheblich ein [Slo05].

Bender hält die Relevanz der statischen Codeanalyse für gering [Ben05]. Dem kann so nicht zugestimmt werden, denn die statische Analyse findet bestimmte Fehler effektiver als die Dynamische. Simon beschreibt in seinem Bericht, dass etwa 20% der identifizierten Fehler nur durch statische Analyse, kaum durch dynamische Tests identifiziert worden wären. Hierzu zählen insbesondere die Bereiche Speicherverwaltung und Initialisierungsanomalien [SS07].

Literatur

[Bal09] Helmut Balzert, *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*, 3 ed., Spektrum Akademischer Verlag, Heidelberg, 2009.

[Bec98] Kent Beck, *Kent Beck's guide to better Smalltalk: A sorted collection*, SIGS reference library series, vol. 14, Cambridge Univ. Press, Cambridge, 1998.

[Ben05] Klaus Bender, *Embedded Systems - qualitätsorientierte Entwickung*, Springer, Berlin, Heidelberg, New York, 2005.

[Bue10a] Frank Buechner, *Fünf Irrtümer über Code Coverage: Fachartikel*, www.elektronikpraxis.vogel.de, 2010.

[Bue10b] ———, *Software-Test: Acht Irrtümer über Code Coverage*, www.elektronikpraxis.vogel.de, 2010.

[GGS08] Michael Goldapp, Ulrich Grottker, and Gregor Snelting, *Validierung softwaregesteuerter Meßsysteme durch Programm Slicing und Constraint Solving*, 2008.

[Hof08] Dirk W. Hoffmann, *Software-Qualität*, eXamen.press, Springer-Verlag, Berlin, Heidelberg, 2008.

[Lig09] Peter Liggesmeyer, *Software-Qualität: Testen, Analysieren und Verifizieren von Software*, 2 ed., Spektrum Akademischer Verlag, Heidelberg, 2009.

[Lis08] Listing Frank, *Software-Testmethoden: Was den Code stark macht*, www.microconsult.de, 2008.

[2]Advanced Simulation and Control Engineering Tool, ist ein Werkzeug für modellbasierte Softwareentwicklung.

[Mat07] The MathWorks, *Code Verification and Run-Time Error Detection Through Abstract Interpretation: A Solution to Today's Embedded Software Testing Challenges*, 2007.

[Mei06] Arne Meier, *Mathematische Beweise und Symbolische Tests*, www.se.uni-hannover.de/lehre/kurz-und-gut.php, 2006.

[Pli09] Julia Pliszka, *Modernisierung von Legacy Anwendungen unter Verwendung von MDA-Konzepten: Bachelorarbeit im Studiengang Angewandte Informatik der Fakultät Technik und Informatik an der HAW-Hamburg*, http://users.informatik.haw-hamburg.de/ ubicomp/projekte/master08-09-aw/pliszka/bericht.pdf, 2009.

[Rei10] Konrad Reiche, *Static Analysis of Embedded Systems: Proseminar: Technische Informatik*, 2010.

[Sch04] Toni Schornböck, *C++ Tutorial: Compiler Fehler*, http://tutorial.schornboeck.net/inhalt.htm, 2004.

[Slo05] Oscar Slotosch, *Erfahrungsbericht Verifikation*, 2005.

[SS07] Daniel Simon and Frank Simon, *Statische Code-Analyse als effiziente Qualitätssicherungsmaßnahme im Umfeld eingebetteter Systeme*, 2007.

[SSC04] Jörn Schneider and Vincent Schulte-Coerne, *Sichere Integration von Fremdsoftware: Lösungskonzepte für eine Problemstellung aus der Praxis*, 2004.

[Wim10] Gernot Wimmer, *Software Engineering: Statische Codeanalyse*, www.gwimmer.info/index.php?id=179, 2010.