Bug Hunting mit statischer Codeanalyse

Roland Bär, Verifysoft Technology, baer@verifysoft.com
Andreas Behr, Verifysoft Technology, behr@verifysoft.com
Daniel Fischer, Hochschule Offenburg, daniel.fischer@hs-offenburg.de

Abstract

Statische Testverfahren zählen zu den analytischen SW-Qualitätssicherungsverfahren. Neben Komplexitätsmetriken, Architektur- und WCET-Analysen stellt insbesondere die statische Codeanalyse eine effiziente Möglichkeit dar, um Software-Bugs zu finden, welche durch dynamische Testverfahren (White-Box- und Black-Box-Testing) nicht entdeckt wurden. Diese unentdeckten Fehler können wie tickende Zeitbomben angesehen werden, die dann im Wirkbetrieb beim Kunden zu schwerwiegenden Systemausfällen führen und damit immense Kosten verursachen können. Neben den klassischen Rules-Checkern (MISRA, CERT) sind insbesondere Ansätze, die datenfluss- und kontrollflussorientierte Verfahren mit interprozeduraler Analyse kombinieren, in der Praxis einfach und effizient einsetzbar. Durch deren globale Sichtweise werden bisher unentdeckte und teils komplexe Bugs im Programmcode gefunden.

1. Einleitung

Fehlerhafte Software kann in vielen Bereichen zu großen Schäden und sogar zum Verlust von Menschenleben führen. Je höher die Kritikalität des (Embedded) Systems, umso höher ist der Aufwand der SW-Qualitätssicherung, die in diesem Umfeld notwendig ist und auch von den Normen sowie Standards eingefordert wird.

In der SW-Qualitätssicherung hat sich die Klassifizierung der einzusetzenden Testverfahren, wie in Abbildung 1 dargestellt ist, bewährt. Während bei den dynamischen Testverfahren das Programm mit dem Ziel ausgeführt wird, die in der Dokumentation spezifizierten funktionalen und nicht funktionalen Anforderungen zu verifizieren, führen statische Testverfahren eine Überprüfung durch, ohne das Programm (Executable) auszuführen. Als Testbasis für die statischen Testverfahren stehen die Dokumente als auch der Sourcecode zur Verfügung. Als strukturierte Gruppenprüfung werden die in der Praxis eingesetzten Prüfverfahren wie Walkthroughs, Reviews, Inspektionen und Audits verstanden, welche sowohl bei Dokumenten als auch bei der Überprüfung des Sourcecodes angewendet werden. Die statische Analyse wird in der Praxis seltener zur Überprüfung der Dokumentation eingesetzt. Der Hauptschwerpunkt dieses Ansatzes ist in der Überprüfung des Programmcodes zu sehen.

Alle aufgezeigten Testverfahren überprüfen letztendlich die Software und deren zugrundeliegenden Spezifikationen nach unterschiedlichen Gesichtspunkten. Sie sind in verschiedenen Phasen des Entwicklungsprozesses verankert und den verschiedenen Teststufen zugeordnet. Gerade in der interdisziplinär ausgerichteten Systementwicklung sind daher auch eine Vielzahl von Abteilungen und externer Dienstleister beteiligt. Hier wird schnell unklar, wer zu welcher Zeit welche Funktionalität in welcher Intensität testet. Die Erstellung eines Master-Testkonzepts (engl. master testplan) in Kombination mit einer neu zu schaffenden Rolle eines

Testmanager kann hier Abhilfe schaffen. Dieser stellt letztendlich sicher, dass die verschiedenen Testverfahren wie z.B. die statische Codeanalyse zielgerichtet zum Einsatz kommen.

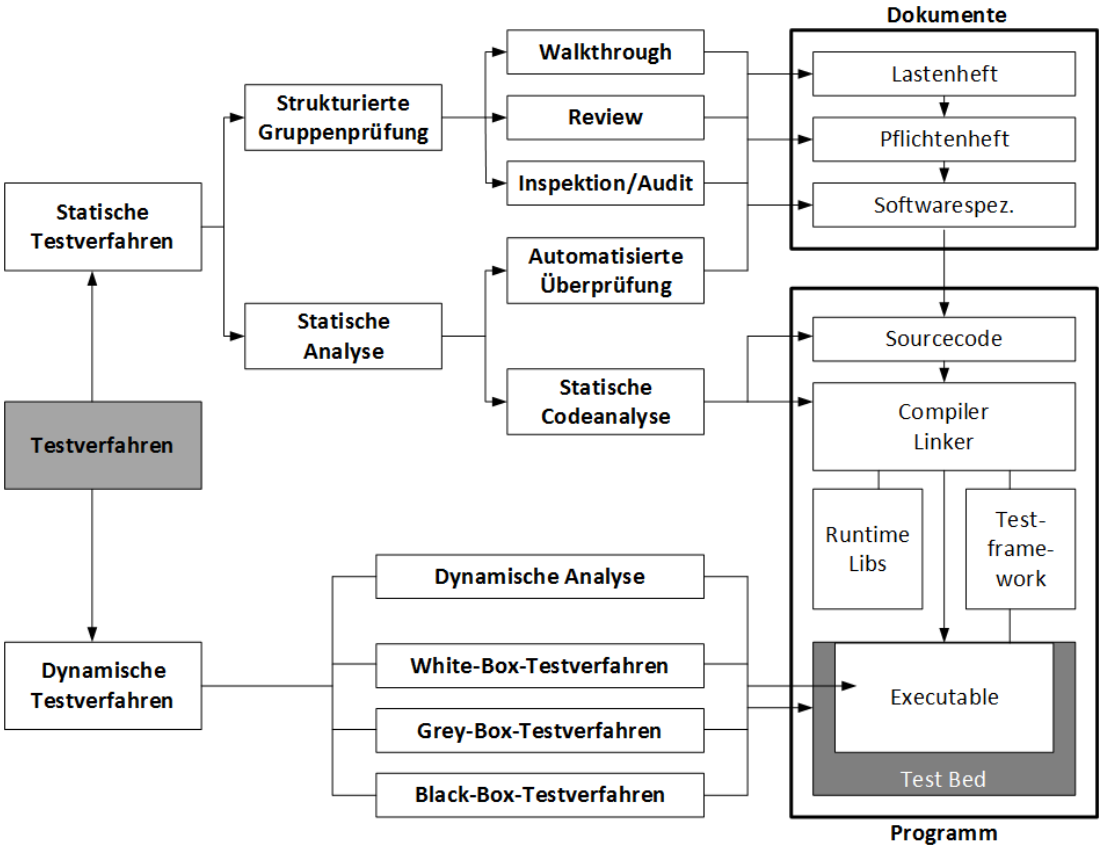


Abbildung 1: *Übersicht Testverfahren*

2. Statische Codeanalyse

In Normen und Standards werden oft die folgenden Varianten der statischen Codeanalyse eingefordert:

- Konformität gegenüber Coding Guidelines
- Rules Checker (MISRA, CERT)
- Kontroll- und Datenflussanalyse
- Analyse und Aufdeckung von Concurreny Fehlern (Race Conditions, …)
- Analyse zur WCET-Bestimmung (Worst Case Execution Time)
- Komplexitätsmetriken (Zyklomatische Komplexität, Lines of Code, …)
- Architekturanalyse (u.a. zum Nachweis der Softwareerosion)

Viele SW-Tools für statische Codeanalyse decken zudem verschiedene Varianten in unterschiedlicher Tiefe und Genauigkeit ab. Daher sollte eine Toolevaluation möglichst strukturiert z.B. nach dem ISTQB-Standard [SL10] durchgeführt werden.

Ein wichtiger Faktor, der oft bei der Toolevaluation kaum berücksichtigt wird, ist die Analysetiefe bei der statischen Codeanalyse. Viele Werkzeuge haben nur eine lokale, auf die jeweilige Prozedur/Funktion ausgerichtete Sichtweise. Diese führen daher nur eine sogenannte intraprozedurale Analyse durch. Leistungsfähigere Werkzeuge

erstellen während der Compilierung ein abstraktes Modell und ermöglichen anhand dieses Modells eine leistungsfähigere interprozedurale Analyse über Funktions- und Modulgrenzen hinweg. Erst dadurch ist es möglich, komplexere und nicht offensichtliche Fehler zu finden. Dies geschieht meist durch eine vollständige Kontroll- und Datenflussanalyse.

Die Datenflussanalyse ist eine leistungsstarke Methode um Anomalien im Datenfluss aufzudecken. Dabei wird jeder lokalen und globalen Variablen einer der drei Zustände zugewiesen:

- u für undefined: Die Variable ist nicht mehr gültig oder es wurde ihr noch kein Wert zugewiesen
- d für defined: Der Variable wurde ein Wert zugewiesen und sie ist gültig
- r für referenced: Auf die Variable wird zugegriffen (z.B. bei einer Berechnung oder bei einem Vergleich)

Verfolgt man nun über die Zeit die Zustände der einzelnen Variablen, so lässt sich dies wie in Abbildung 2 aufgezeigt ist, als Stränge von Zuständen darstellen.

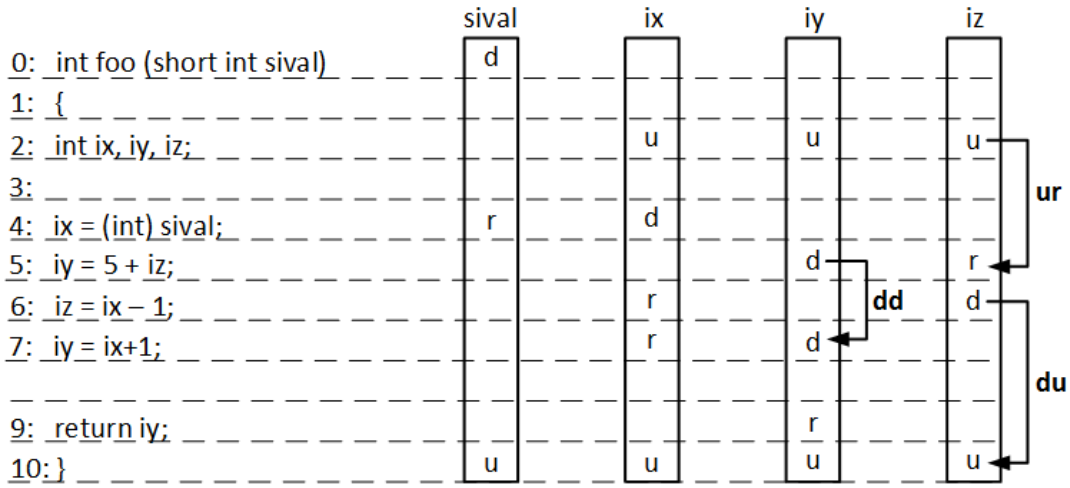


Abbildung 2: *udr-Zustände der Variablen sival, ix, iy und iz sowie deren Anomalien*

Befinden sich nun in einem Strang nacheinander die Zustände dd, ur oder du, so spricht man von Anomalien (dd-, ur- sowie du-Anomalie). Diese weisen auf einen möglichen Fehler hin. Enthält eine Funktion Schleifen und Abfragen, so teilen sich die Stränge für jede Variable entsprechend auf. Listing 1 zeigt auf, dass nur die gleichzeitige Kombination von Kontroll- und Datenfluss (Beispiel angepasst aus [L2]) sinnvoll ist. Nur unter Berücksichtigung des Kontrollflusses (hier zwei if-Abfragen) fällt auf, dass die malloc-Anweisung nie in Kombination mit der notwendigen free-Anweisung auftreten kann.

```
void goo (int i)
{
int *p;

if (i==3)
   p = (int*) malloc(10*sizeof(int));

aSubroutine(p);

if (i==10)
```

```
   free(p);
}
```

Listung 1: *Kombination von Kontroll- und Datenfluss*

Entsprechend komplex wird es, wenn Variablen per Referenz (siehe Funktionsaufruf aSubroutine in Listing 1) übergeben werden oder auf globale Variablen in unterschiedlichen Funktionen zugegriffen wird. Sollen auch diese Variablen einer kontroll- und datenflussorientierten Analyse unterzogen werden, so bedarf es einer interprozeduralen Analyse.

Gerade der Umgang mit Zeigern über Funktionsgrenzen hinweg und unter Einbeziehung komplexer Kontrollflüsse (messbar mit der Metrik Cyclomatic Complexity) ist sehr fehleranfällig und kann nur von leistungsstarken Werkzeugen entdeckt werden.

3. Nutzen der statischen Codeanalyse

Im Gegensatz zu den White-Box-, Grey-Box- und Black-Box-Testverfahren bietet die statische Codeanalyse einen grundlegenden Vorteil. Es sind keine Testfälle und Testprozeduren zu implementieren. Es fallen lediglich ein Installationsaufwand sowie ein geringer Konfigurationsaufwand an, um diese Werkzeuge in die bestehende Toolkette zu integrieren.

Diese Werkzeuge können nun als schritthaltende QS-Maßnahme in der Implementierungsphase (Development Testing) oder in der Test- und Wartungsphase nach den jeweiligen Bugfixes zur Anwendung kommen. In [YZPS11] wurde in Untersuchungen basierend auf der Entwicklung von vier Betriebssystemen (OpenSolaris, FreeBSD, Linux sowie ein kommerzielles System) aufgezeigt, dass

- 14.8-24.4% aller Postrelease Bugfixes fügen einen neuen Fehler ein
- 39% der Concurrency Bugfixes inkorrekt sind
- Bugfixes häufig von Entwicklern gemacht werden, die den Code nicht kennen. Deren Bugfixes führen dann häufiger zu neuen Fehlern

Meist erfolgen gerade in der Wartungsphase keine umfassenden und gründlichen Nachtests. Dies kann dann dazu führen, dass vermeintlich verbesserter Code letztendlich mit noch schwerwiegenderen Bugs beim Anwender landet. Dieses Szenario kann zumindest mit statischer Codeanalyse entschärft werden.

4. Exemplarische Beispiele

Es soll nun anhand der Werkzeuge Cppcheck, PC-lint und CodeSonar exemplarisch aufgezeigt werden, welche Fehler gefunden werden. Bei Cppcheck handelt es sich um ein Opensource Projekt ([L2]), während es sich bei PC-lint ([L3]) und CodeSonar ([L1]) um kommerzielle Produkte handelt. Die Hauptausrichtung von PC-lint ist die Überprüfung der MISRA-Regeln, während CodeSonar sich auf ein vollständige interprozedurale Kontroll- und Datenflussanalyse konzentriert. Ebenso werden bei CodeSonar Komplexitätsmetriken und eine Visualisierung der Softwarearchitektur zur Vermeidung von Softwareerosion unterstützt.

```
void bufferoverrun ()
{
char buffer[10];
int i,j;
char * pc;

/*buffer overrun 1 */
buffer[10] = 'a';

/* buffer overrun 2 */
for (i=0; i<=10;i++)
   buffer[i] = 'b';

/* buffer overrun 3 */
pc = &buffer[0];
for (i=0;i<=10;i++)
   *pc++= 'c';
j=0;
for (i=0;i<6;i++)
   buffer[j++]= 'd';

/* buffer overrun 4 */
for (i=0;i<6;i++)
   buffer[j++]= 'e';
}

int notinit (int k)
{
char * pc1;
int iret;
/* not initialized 1*/
*pc1 = 'a';
if (k > 42)
   iret = 1;
/* not initialized 2*/
return iret;
}

void dead_code(void)
{
int x = some_func();
int y = some_other_func();
if (x)
{
   if (y>10 && !x)
   {
      x++;   /* dead code*/
      return;
   }
}
}
```

```
int simplememleak()
{
   char *p = (char*) malloc(12);
   if (!p)
   {
      return 0;
   }

   if (!some_func())
   {
      free(p);
      return -1;
   }

   if (!some_other_func())
   {
      return -2; /* memory leak */
   }
   free(p);
   return 1;
 }

void simpleuseafterfree()
{
char *pc1;
char *pc2;
char *pc3;

pc1 = (char*) malloc(10*sizeof(char));
if (pc1)
{
   pc1[0] = 'a';
   free(pc1);
   pc1[0] = 'b';/*use after free 1*/
}

pc2 = (char*) malloc(10*sizeof(char));
if (pc2)
{
   pc3 = pc2;
   free(pc2);
   pc3[0] = 'b';/*use after free 2*/
}
}
```

Listing 2: *Exemplarische Fehler (intraprozedural)*

In Listing 2 ist exemplarisch fehlerhafter Programmcode dargestellt. Die jeweiligen intraprozeduralen Fehler sind durch einen Kommentar markiert. Dieser Programmcode wurde einer statischen Codeanalyse mit Cppcheck, PC-lint und CodeSonar unterzogen. Die Ergebnisse dieser statischen Codeanalyse sind in Tabelle 1 dargestellt.

Exemplarischer Fehler	Cppcheck	PC-lint	CodeSonar
buffer overrun 1	ja	ja	ja
buffer overrun 2	ja	ja	ja
buffer overrun 3			ja
buffer overrun 4			ja
not initialized 1	ja	ja	ja
not initialized 2	ja	ja	ja
dead code			ja
memory leak	ja	ja	ja
use after free 1	ja	ja	ja
use after free 2			ja

Tabelle 1: *Erkannte exemplarische Fehler bei intraprozeduraler Codeanalyse*

Bei einer interprozeduralen Analyse über Modulgrenzen hinweg, ergibt sich ein völlig anderes Bild. In Listing 3 sind exemplarische Funktionen dargestellt, die sich in zwei C-Dateien befinden.

Modul1.c

```
void testdriver ()
{
int *p1;
int *p2;
int *pn = NULL;
int *pnotinit;
int test[4];

p1 = (int*) malloc
(1*sizeof(int));
p2 = (int*) malloc
(10*sizeof(int));

test_free(p2,10);

if (p1)
   test_buffer_overrun(p1,
test);

if (p1)
   free (p1);/*double free*/

test_use_afterfree(p1);

test_pointer(pn, pnotinit);
}
```

Modul2.c

```
void test_buffer_overrun(int* p, int
test[])
{
int * pdum;
test[3] = 4;
test[4] = 5;/*buffer overrun stat.*/
if (p)
{
   pdum = p+1;
   *(pdum)=42;/*buffer overrun dyn.*/
   free(p); /*double free*/
}
}

void test_pointer(int * pn, int *
pnotinit)
{
if (pnotinit)
   *pnotinit=42;/*pointer uninit.*/

*pn = 42;/*NULL pointer deref.*/
}

void test_free(int *p, int x)
{
if (p && x < 10) /*memory leak*/
   free (p);
}

void test_use_afterfree(int *p)
{
   *p=42; /*use after free*/
}
```

Listing 3: *Exemplarische Fehler (interprozedural)*

Es zeigt sich, dass hier CodeSonar über Modulgrenzen hinweg Fehler erkennt, während die anderen untersuchten Tools hier nur in Ansätzen auf mögliche Fehler hinweisen (Tabelle 2).

Exemplarischer Fehler	**Cppcheck**	**PC-lint**	**CodeSonar**
double free			ja
buffer overrun (statically)			ja
buffer overrun (dynamically)			ja
pointer uninitialized		(Ja) intraprozedural	ja
NULL pointer dereference			ja
memory leak			ja
use after free		(Ja) intraprozedural	ja

Tabelle 2: *Erkannte exemplarische Fehler bei interprozeduraler Codeanalyse über Modulgrenzen hinweg*

PC-lint erkennt den nicht initialisierten Pointer nur in Modul1.c, ebenso wird intraprozedural für Modul1.c erkannt, dass nach free(p1) erneut p1 verwendet wird. In Modul2.c werden keine Fehler erkannt. Cppcheck findet keinen der exemplarischen Fehler.
Es ist für den Anwender nützlich, wenn aufgezeigt wird, über welche Pfade und unter welchen Bedingungen (Events) es zu einem Fehler kommt. Dies ist beispielhaft für den obigen Fehler „memory leak“ (Funktion test_free()) in Abbildung 3 aufgezeigt: Bedingt durch die Vorgeschichte im gewählten Pfad, wird in Event5 die if-Abfrage als false evaluiert und es erfolgt daher kein free. Ohne diese Information (interprozeduraler Kontroll- und Datenfluss) ist das Auffinden der Ursachen für das Entstehen eines Bugs recht aufwändig und fehlerträchtig.

5. Zusammenfassung

Die statische Codeanalyse zählt zu den statischen Testverfahren und kann zeitnah (Development Testing) während der Implementierung Bugs im Code entdecken. In der Test- und Wartungsphase stellt die statische Codeanalyse sicher, dass keine neuen Fehler in den Code eingefügt werden. Es gibt unterschiedliche Arten von Werkzeugen zur statischen Codeanalyse. Diese unterscheiden sich in Funktionsumfang und Tiefe der Analyse. Weiterhin empfiehlt es sich, verschiedene Werkzeuge zur statischen Codeanalyse einzusetzen, da diese unterschiedliche Ausrichtungen und Funktionalitäten haben. Eines haben alle Werkzeuge gemeinsam: Es sind keine Testfälle zu spezifizieren und zu implementieren. Nach der Installation kann man gleich mit dem Bug-Hunting beginnen.

Abbildung 3: *Darstellung der Pfade und Events am Beispiel des Fehlers memory leak mit dem Tool CodeSonar*

Literatur

[ALSU08] Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi; Ullman, Jeffrey D.: *Compiler – Prinzipien, Techniken und Werkzeuge*, 2. Auflage, Pearson Studium, 2008

[YZPS11] Yin, Zuoning; Zhou, Yuanyuan; Pasupathy, Shankar; Bairavasundaram, Lakshmi: *How do Fixes Become Bugs?*, Proceedings, 19th ACM SIGSOFT symposium, Hungary, 2011

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

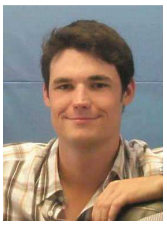
[SL10] Spillner, Andreas; Linz, Thilo: *Basiswissen Softwaretest*, dpunkt Verlag, 4. Auflage, 2010

Links

[L1] http://www.verifysoft.de/de_grammatech_codesonar.html
[L2] http://cppcheck.sourceforge.net/
[L3] http://www.gimpel.com/html/pcl.htm

Autoren

Roland Bär leitet die Entwicklung und den Support bei Verifysoft Technology. Schwerpunkte seiner Tätigkeit sind die statische Codeanalyse (Produkt CodeSonar) sowie die Weiterentwicklung und Erweiterung des Coveragetools Testwell CTC++ speziell für kleine Targets. Compilerspezifische Anpassungen der Werkzeuge gehören ebenso zu seinem Fachgebiet.



Andreas Behr ist bei Verifysoft Technology u.a. für die Integration von Testwell CTC++ in verschiedene Entwicklungsumgebungen zuständig. Ebenso ist er für die nationalen und internationalen Produktschulungen von Testwell CTC++ verantwortlich.

Daniel Fischer ist seit 2001 Professor für Informationstechnologie an der Hochschule Offenburg. In Zusammenarbeit mit der Firma Verifysoft Technology bietet er seit geraumer Zeit die Seminare „Testen von Embedded Software“ und „Effizientes Testmanagement“ an. Seit 2013 ist er ISTQB Certified Tester Full Advanced (Technical Test Analyst, Test Analyst und Test Manager).