4 Automatische statische Code-Analyse

Die Software-Code-Review, also die Analyse des Quellcodes durch eine vom Programmautor verschiedene Person, ist eine der wichtigsten Verifikationsmaßnahmen in der Software-Entwicklung mit hohem Integritätsanspruch. Zwischen 30% und 50% aller Programmfehler werden vor der ersten Ausführung gefunden, zumindest bei »klassischen« Programmiersprachen [Fagan 86]. Leider benötigt eine gute Review aber viel Zeit. Einige Software-Werkzeuge versprechen, automatisch in wenigen Sekunden Reviews durchzuführen. Dieses Kapitel diskutiert, was von solchen Versprechungen zu halten ist, und erläutert (beispielhaft an der Programmiersprache C), wie diese Werkzeuge funktionieren.

4.1 Motivation zum Einsatz von Analysewerkzeugen

Kostet Sie ein Programmfehler viel Geld, so werden Sie nach dieser Erfahrung in zukünftigen Code-Reviews den Quellcode dahingehend prüfen, dass so ein Fehler nicht nochmals unbemerkt das Haus verlässt. Damit Kollegen auch von Ihrer Erfahrung profitieren, erfassen Sie vielleicht solche Fehler in einer Liste. In zukünftigen Code-Reviews wird dann die Liste vom jeweiligen Reviewer zur Hand genommen. Je länger die Liste, desto besser für die Qualität der Review, desto langweiliger für den Reviewer und desto teurer, weil zeitaufwändiger.

Die Idee, im Quellcode nach Fehlermustern mit einem Werkzeug automatisch zu suchen, liegt also fast auf der Hand und ist daher denkbar alt: *Lint*, ein Hilfsprogramm aus der Unix-Werkzeugkiste, tut genau das seit Ende der 1970er. Aus dem Lint-Handbuchtext: »Lint versucht Eigenheiten eines C-Programms zu entdecken, die aller Wahrscheinlichkeit nach Fehler sind, nicht portierbar sind oder unnötig. Es prüft die Typen von Variablen strikter als viele Compiler, findet unerreichbare Anweisungen, unbenutzte Variablen und logische Ausdrücke, die einer Konstanten entsprechen. Darüber hinaus prüft Lint, ob die Rückgabewerte von Funktionen auch tatsächlich verwendet werden.«

Basierend auf der Idee von Lint existieren heute Werkzeuge, die Quellcode nach verdächtigen Programmzeilen durchsuchen und den Programmierer warnen, wenn sie fündig werden. Diese Werkzeuge existieren längst nicht mehr für C

alleine, sondern für eine große Zahl von Programmiersprachen. Das Spektrum an Werkzeugen reicht von Freeware über etwas mächtigere kommerzielle Programme im Verkaufswert von 100 bis 1000 Euro bis hin zu High-End-Produkten, teilweise weit jenseits der 5000 Euro. Produkte im Hochpreissegment prüfen häufig auch einen, zumindest zum Teil, frei definierbaren Software-Coding-Standard.[1]

Was diese Tools für uns tun, wird *automatische statische Code-Analyse* genannt. Statisch deshalb, weil die zu prüfende Software dabei nicht ausgeführt wird. Zur statischen Analyse gehört auch das Erheben von Code-Metriken. Diese Maßzahlen versuchen, eine gewisse Aussage über die Qualität von Quellcode zu geben.

Die folgenden drei Unterkapitel widmen sich dem Thema Bugfinding durch Analysewerkzeuge, der darauf folgende Abschnitt 4.5 beschäftigt sich dann mit der Erhebung der besagten Metriken.

4.2 Techniken von Analysewerkzeugen im unteren Preissegment

Wie eingangs erwähnt, ist das Prüfen jeder Programmzeile nach einer Liste von potenziellen Fehlermustern eine ermüdende Angelegenheit. Speziell »kleine« Fehler, wie eine eventuell ungewollte implizite Typenkonversion werden leicht übersehen. Genau dies ist aber die Stärke einer automatischen Analyse. Die Nachfolger von Lint ermüden auch nach tausenden Codezeilen nicht und decken solche Gefahrenstellen auf. Listing 4–1 gibt eine Kostprobe. Das C-Programm ist voll von Fehlern, die automatisch erkannt werden können.

```
/* Programmiersprache C */

#include <string.h>

char* MyName();                                  /* 01 */
char cUartData = (char) 0xFA;                    /* 02 */


main()
{
   unsigned uiA = (unsigned) cUartData;          /* 03 */
   int i, a[10] = {0,0,0,0,0,0,0,0,0,0};         /* 04 */
```

1. Ein Software-Coding-Standard ist eine Vereinbarung zur Verwendung einer Programmiersprache. So ein Standard definiert zum Beispiel das Verbot der Verwendung bestimmter Sprach-Konstrukte (wie `goto` und `longjmp`) oder die Einhaltung bestimmter Richtlinien zur Namensgebung von Variablen. Die wohl bekanntesten Coding-Standards sind von der Motor Software Reliability Association (MISRA) für die Programmiersprachen C und C++. Nachdem diese Coding-Standards weit verbreitet sind, gibt es auch sehr kostengünstige Werkzeuge, die die Einhaltung der MISRA-Regeln überprüfen.

```
for (i = 0; i <= 10; i++)                    /* 05 */
   {
      a[i] += i;                             /* 06 */
   }
   for (i = 0; i <= 3; i++);                 /* 07 */
   {
      char szString[10];                     /* 08 */
      a[i] -= cUartData;                     /* 09 */
      strcpy(szString, MyName());            /* 10 */
   }
}


char* MyName(int i)
{
   char name[11] = "Joe Jakeson";            /* 11 */
   if (i) name[2] = 'i';                     /* 12 */
   return name;                              /* 13 */
}
```

Listing 4–1 *Code zur Demonstration eines einfachen Analysewerkzeugs*

Ein leistungsfähiges kommerzielles Programm zur statischen Code-Analyse hat am Programm in Listing 4–1 einiges auszusetzen: In Zeile 1 wird zu Recht gewarnt, dass die Deklaration des Prototyps der Funktion `MyName()` unvollständig ist. Für einen C++-Compiler ist in Zeile 1 das Parameterprofil für `MyName()` genau definiert: keine Parameter, also `void`. In C ist dem nicht so: `void` muss explizit in Klammern stehen, wenn die Funktion keine Parameter erwartet. Bei einer Deklaration wie in Zeile 1 sind die Parameter der Funktion undefiniert und der C-Compiler muss eine Annahme machen, sobald er einen Funktionsaufruf übersetzen muss. Nicht alle Compiler warnen den Benutzer, wenn sie so eine Annahme treffen müssen.

Auch Zeile 3 ist tückisch. Ein gutes Analysewerkzeug würde hier einen verdächtigen Type-Cast melden. Bei der Umwandlung des Typs `char` in den Typ `unsigned` wird nämlich zuerst implizit `char` in `int` gewandelt. Bei Architekturen, bei denen der Typ `char` vorzeichenbehaftet ist, wird dabei das in diesem Fall gesetzte Vorzeichenbit von `cUartData` auf die neue Bitbreite erweitert. Die Variable `uiA` enthält dann zum Beispiel den Wert `0xFFFFFFFA` und nicht `0xFA`, wie es der Programmierer vermutlich wollte.

Zählen Sie bei Reviews von Feld-Initialisierungen wie in Zeile 4 immer nach, ob tatsächlich die richtige Anzahl von Variablen initialisiert ist? Ein Analysewerkzeug tut das. In Zeile 4 fehlt eine Null. Das Werkzeug kann sogar Zugriffe mit unerlaubten Indizes erkennen, sofern dies durch Auswertung der Zahlenliterale und Konstanten möglich ist. In Listing 4–1 ist das der Fall. In Zeile 6 werden die Indexgrenzen einer Feldvariablen überschritten und diese Überschreitung kann automatisch erkannt werden.

In Zeile 7 sehen wir einen Fehler, dessen Erkennung für Lint-Derivate ein leichtes Spiel ist, der aber trotzdem häufig vorkommt und bei manueller Inspek-

tion nicht so schnell auffällt: Der Strichpunkt am Ende der Schleifenanweisung ist offensichtlich nicht gewollt.

Die Genauigkeit und Pedanterie einer programmunterstützten Analyse von Quellcode raubt manchen Programmierern viel Energie, wenn sie erstmals ein penibles Werkzeug an bislang ungeprüftem Quellcode ausprobieren. Auch in Listing 4–1 ist ein Fehler enthalten, der keine Auswirkung hat und dennoch von einem Analyseprogramm bemängelt würde: Der Rückgabetyp der Funktion `main()` ist nicht explizit definiert. Gemäß C-Standard muss der Compiler daher annehmen, dass der Rückgabewert vom Typ `int` ist. In `main()` ist aber keine `return`-Anweisung zu finden. Die Praxis, den Rückgabetyp des Hauptprogramms nicht explizit anzugeben, wenn `main()` keinen Wert an das Betriebssystem liefert, ist aber weit verbreitet. Schon beim berühmten »hello world«-Programm von den C-Veteranen Kernighan und Ritchie, in Listing 4–2 gezeigt, ist das so. Streng genommen hat dieses Programm nach heutigen C-Standards aber zwei Fehler [Stroustrup 02].

```
main()
{
   printf("Hello, world\n");
}
```

Listing 4–2 *Das Hello-World-Programm*

Analysewerkzeuge im unteren Preissegment neigen dazu, den Benutzer mit unnötigen Warnungen zu überhäufen. Als Abhilfe können in der Regel Warnungen Zeile für Zeile durch spezielle Kommentare unterdrückt oder global durch Setzen eines Parameters ausgeschaltet werden.

4.2.1 Sprachspezifische Fallstricke

Die Programmiersprache C spannt einige Fallstricke für den Programmierer, die die Portierbarkeit von Code beeinträchtigen. Da wären unter anderem die undefinierte Auswertungsreihenfolge bei arithmetischen Ausdrücken, compilerabhängige Ergebnisse beim Rechts-Shift von negativen ganzen Zahlen und die undefinierte Bitbreite des Datentyps `int`. Nach diesen Stolperfallen sucht eine automatische Analyse im Code. Auch die unsichtbare Nullterminierung von Strings hat schon so manchen Programmfehler ausgelöst. Über diesen Fallstrick stolpert übrigens Zeile 11 von Listing 4–1: Die Feldvariable `name` reserviert nicht genug Speicher, um die Endmarkierung `'\0'` zu speichern.

Der letzte, automatisch erkennbare Fehler von Listing 4–1 ist in Zeile 13 zu sehen. Es wird versucht, als Rückgabewert eine Variable zu verwenden, die mit dem Verlassen der Funktion nicht mehr existiert. Dies ist eine Art von Fehler, die auch von vielen Compilern gemeldet wird.

4.2.2 Kontrollflussanalyse

Die automatische Überprüfung von Quellcode macht es auch möglich, toten Code und unbenutzte Variablen auszuforschen. Speziell bei der Übernahme von Code aus anderen Projekten kann das sehr hilfreich sein. In Listing 4–1 gibt es übrigens zwei Variablen, die niemals referenziert werden.

Die *Kontrollflussanalyse* genannte Analysetechnik kann aber mehr, als nur toten Code aufzuspüren. Es werden die verschiedenen möglichen Ausführungspfade auf eine Reihe von potenziellen Unstimmigkeiten untersucht. Die meisten Tools erstellen für die Suche nach Kontrollflussanomalien einen Kontrollflussgraphen. Die Knoten des Kontrollflussgraphen sind Basisblöcke; das sind Code-Abschnitte, in die die Flusssteuerung an genau einem Einstiegspunkt eintritt und ohne möglichen Halt oder Verzweigung nur an genau einem Punkt wieder austreten kann [Pezzé 09]. Zwei typische Warnungen, die aufgrund der Analyse dieser Graphen ausgegeben werden, zeigt Listing 4–3.

```
/* Programmiersprache C */

#include <stdio.h>
extern int foo(void);

int sign(int x)
{
   if (x>=0)
   {
      return 1;
   }
   else
   {
      return -1;
   }
   return 0; /* Warnung: unerreichbarer Code */
}

int main(String args[])
{
   int i = foo();
   switch (i)
   {
      case 1: printf("Fall 1");
      break;
      case 2: printf("%d", sign(12));
      break;
      case 3: printf("%d", sign(-8));
      case 4: printf("Fall 4");
      break;
   }
   /* Warnung: ein case-Statement ohne break */
   return 0;
}
```

Listing 4–3 *Warnung bei Kontrollflussanomalien*

Während die erste Warnung definitiv auf Unsinn hinweist, kann die zweite Warnung durchaus ein Falschalarm sein und sollte in diesem Fall im Werkzeug *für diese spezielle Zeile* abgeschaltet werden, damit man nicht bei jedem neuerlichen Check damit belästigt wird. Es ist auch gute Praxis, im Code durch einen Kommentar darauf hinzuweisen, wenn man absichtlich einen `case`-Block ohne `break`-Statement beendet.

4.2.3 Datenflussanalyse, Initialisation Tracking

Das Beispiel in Listing 4–1 demonstriert eine Reihe von Fähigkeiten der automatischen Code-Analyse, bei Weitem aber noch nicht alle, zum Beispiel die Suche nach der Verwendung uninitialisierter Variablen. Im einfachsten Fall wird dabei geprüft, ob im Quellcode für jede Variable vor ihrer ersten Verwendung eine Zuweisung existiert. Listing 4–4 zeigt ein Programm, das mit dieser einfachen Technik nicht ausreichend geprüft wird: Den Variablen a, b und c wird zwar ein Wert zugewiesen, die Zuweisungen sind aber an Bedingungen geknüpft.

```
/* Programmiersprache C */

void oje(void)
{
   int a,b,c,m,n,p;

   scanf("%d", &a);

   if (a) b = 6; else c = b;
   a = c;

   while (n--)
   {
      /* ... */
      m = 6;
      /* ... */
   }
   p = m;
}
```

Listing 4–4 *Jede Menge uninitialisierte Variablen*

Auch wenn das Programm syntaktisch korrekt ist, warnen die meisten Compiler bei einer Verwendung von uninitialisierten Variablen. Doch häufig ist das oben beschriebene einfache Verfahren implementiert, das bei Listing 4–4 nicht greift. Während ein Compiler also in Listing 4–4 möglicherweise *keine* Fehlerquelle meldet, findet ein statisches Code-Analyseprogramm *vier* Stellen, wo auf eine uninitialisierte Variable zugegriffen wird: `b` wurde nicht initialisiert, `c` ist wahrscheinlich nicht initialisiert, `n` ist nicht initialisiert, `m` ist wahrscheinlich nicht initialisiert. Der Grund für die Überlegenheit der Spezialsoftware gegenüber der Überprüfung des Compilers liegt im intelligenteren Algorithmus, der Bedingungen verfolgt, und in

der höheren Anzahl der Durchläufe, mit der der Code analysiert wird. Das Prüfprogramm verfolgt, wie sich ein Fehler fortpflanzt.

Beim Schreiben dieses Texts hat der Buchautor zwei Compiler an Listing 4–4 versucht. Die Verwendung von `n` als nicht initialisierter Schleifenzähler wurde von beiden Compilern nicht erkannt. Der Grund dafür ist, dass nicht `n` selbst, sondern ein arithmetischer Ausdruck als Schleifenbedingung gewählt wurde. Ein gutes Analyseprogramm lässt sich davon nicht irritieren und erkennt auch, dass durch die Schleifenbedingung die Variable `m` möglicherweise nie initialisiert wird.

4.2.4 Datenflussanalyse, Value Tracking

Eine weitere Möglichkeit Fehler durch das Verfolgen von Zahlenwerten im Quellcode zu finden, ist die Analyse von Indizes von Feldvariablen sowie das Erkennen der Dereferenzierung eines Null-Zeigers und von Divisionen durch Null. Im Gegensatz zu den bisherigen Beispielen im vorherigen Abschnitt 4.2.3 beschränkt sich diese Analyse nicht auf Initialisierungswerte alleine. Listing 4–5 zeigt einige Beispiele. In der ersten Funktion kann ein intelligentes Analyseprogramm aus der Bedingung der `if`-Anweisung erkennen, dass die Variable `k` größer oder gleich 10 sein könnte. Die Verwendung als Index für eine Feldvariable mit der Dimension 10 führt daher zu einer Warnung.

In der zweiten Funktion passiert Ähnliches. Die Bedingung der `if`-Anweisung lässt den Schluss zu, dass die Zeigervariable auf Null zeigen kann und daher die beiden nachfolgenden Anweisungen Probleme bescheren können.

```
/* Programmiersprache C */

int a[10];

int eins(int k, int n)
{
   if (k >= 10) a[0] = n;
   return a[k];           /* Warnung: k könnte 10 sein */
}

int *zwei(int *p)
{
   if (p) printf("\n");   /* p könnte NULL sein ... */
   printf("%d", *p);      /* Warnung */
   return p + 2;          /* Warnung */
}

void drei(void)
{
   int a[10], *p, *q;
   p = a + 10;            /* OK, nicht unüblich */
   *p = 0;                /* Warnung */
   q = p + 1;             /* Warnung */
   q[0] = 0;              /* Warnung */
}
```

```
int x(int n)
{
   return n - 1;
}

int y(int n)
{
   return n / x(1);       /* Warnung nach 2. Durchlauf */
}

int z(int n)
{
   if (n > 0) n = 0;
   else if (n <= 0) n = -1;   /* Warnung */
   return n;
}
```

Listing 4–5 *Beispiele für automatisch erkennbare, datenabhängige Programmierfehler*

Eine gute Analyse wertet arithmetische Ausdrücke mit Zeigervariablen aus, sofern dies möglich ist. Somit wird in der dritten Funktion erkannt, dass bei den zwei in Listing 4–5 gezeigten Zuweisungen je eine Speicherzugriffsverletzung stattfindet beziehungsweise der Zeiger `q` nach der letzten Zuweisung auf eine undefinierte Stelle im Speicher zeigt.

Als vorletztes Beispiel zum Thema Value Tracking zeigt Listing 4–5 die beiden Funktionen `x()` und `y()`, wo zu sehen ist, warum ein Werkzeug zur automatischen Code-Analyse den Quelltext eines Programms in mehreren Durchläufen analysiert, wie zuvor erwähnt. Im ersten Durchlauf lernt das Analyseprogramm, dass `x()` mit dem Argument 1 aufgerufen wird. Im zweiten Durchlauf wird, wenn die Definition von `x()` erneut verarbeitet wird, gefolgert, dass dieses Argument einen Funktionsrückgabewert von Null zur Folge hat. Wenn im zweiten Durchlauf dann `y()` abgearbeitet wird, wird die Gefahr der Division durch Null erkannt und gemeldet.

Beachten Sie, dass das letzte Beispiel die Auswertung der Funktion erfordert und eine funktionsübergreifende Analyse darstellt. In manchen Analyseprogrammen kann sogar die Anzahl der Durchläufe eingestellt werden. Mit einer höheren Anzahl an Durchläufen arbeitet sich das Werkzeug tiefer in einen möglichen Aufrufbaum. Damit ist die Chance größer, auch bei Rekursionen noch Probleme zu erkennen.

Als kleine Zugabe erlaubt Value Tracking auch unsinnige Bedingungen zu erkennen, wie die Warnung zur zweiten `if`-Bedingung der Funktion `z()` in Listing 4–5 zeigt.

4.2.5 Semantische Analyse

Auch Analysewerkzeuge aus dem unteren Preissegment wissen über die Semantik von Funktionsparametern und Rückgabewerten von Funktionen der Standardbi-

bliothek Bescheid. Mit diesem Wissen kann die zuvor vorgestellte Technik erweitert werden, um Plausibilitätstests für die Parameterübergabe durchzuführen. Ein Beispiel dazu ist in Listing 4–6 zu sehen. Warnung 1 kann nur ausgegeben werden, weil das Analyseprogramm weiß, dass `fopen()` eine spezielle Funktion ist, deren erstes Argument nicht Null sein darf. Die Analyse des Ausdrucks vor dem Funktionsaufruf zeigt aber, dass sie Null sein könnte, sonst hätte die Abfrage auf Null keinen Sinn.

Warnung 2 wird gemeldet, weil über `fopen()` noch eine zweite semantische Eigenheit bekannt ist: Der Rückgabewert kann Null sein. Daher könnte `fclose()` auch mit einem ungültigen Parameterwert aufgerufen werden.

In Listing 4–6 ist im Kommentar auch der Prototyp der Funktion `fread()` zu sehen. Zusätzlich zur Prüfe-auf-Null-Semantik für das zweite und dritte Argument können Werkzeuge hier einen komplexeren Test ausführen: Wenn die Größe des zweiten Parameters multipliziert mit dem dritten Parameter die Größe des als ersten Parameter übergebenen Puffers übersteigt, dann wird ein Fehler gemeldet. Das ist im Listing als Warnung 3 markiert. Moderne Werkzeuge haben eine ganze Reihe von Semantiken für Standardfunktionen hinterlegt und gestatten auch die Definition von Semantiken für selbst geschriebene Funktionen.

Noch ein paar Beispiele für mögliche semantische Prüfungen für Funktionen der C-Standard-Bibliothek:

- Ein Aufruf der Funktionen `abort()` und `exit()` kehrt nicht mehr zurück. Danach stehender Code ist potenziell toter Code.
- Für `fgets()` wird geprüft, ob die Größe des zweiten Parameters die Puffergröße des ersten überschreitet.
- Der Zeiger, der der Funktion `free()` übergeben wird, ist nach dem Funktionsaufruf uninitialisiert und darf daher nicht dereferenziert werden.
- Nach dem Aufruf von `fclose()` ist das Argument der Funktion ein uninitialisierter Zeiger.

```
/* Beispiel semantischer Fehler in einem C-Programm */

void semantik(char *name)
{
   char buf[100];
   FILE *f;

   if (name == 0) printf("ok\n");
   f = fopen(name, "r"); /* Warnung 1: name könnte 0 sein */

   /* char *fread(char *, size_t, size_t, FILE *); */
   fread(buf, 100, 2, f);                  /* Warnung 3 */

   fclose(f);          /* Warnung 2: f könnte NULL sein */

}
```

Listing 4–6 *Beispiele für Fehler, die mithilfe semantischer Regeln entdeckt werden*

4.2.6 Starke Typenprüfung

Die Programmiersprache C erlaubt zwar die Definition von Typen als Basistypen (wie etwa Typ Strom als `float`, Typ Spannung als `double`), doch bei der Typenprüfung beschränkt sich der C-Compiler auf die Basistypen anstelle der Eigendefinitionen. Somit würden in diesem Beispiel Volt und Ampere ohne jede Warnung addiert werden. Deshalb werden die nur mit `typedef` definierten Typen auch schwache Typen genannt. Wären Strom und Spannung aus dem vorherigen Beispiel starke Typen, so wäre eine Addition der beiden Größen nur mit Typenkonversion möglich.

Für den Einsatz von Software in Geräten mit Sicherheitsrelevanz empfiehlt die Norm EN 61508 dringend die Verwendung einer Sprache mit starker Typenprüfung. C ist zwar für eingebettete Systeme häufig im Einsatz, doch C unterstützt die starke Typenprüfung nicht. Damit C dem Stand der Technik entsprechend für sicherheitsrelevante Geräte eingesetzt werden kann, muss daher ein Analyse-Tool verwendet werden, das dieses Manko kompensiert und eine starke Typenprüfung anbietet.

Erfahrungsbericht mit Statischer Code-Analyse

Ich habe für ein Unternehmen getestet, das Firmware in C für einen Mikrocontroller mit 16 KB ROM entworfen hat. Die Firmware war bereits in mehreren hundert Geräten am Markt und gehorchte keinem Coding-Standard, sondern war ein eher hässlicher Hack. Im Rahmen einer Produkterweiterung setzten wir PC-Lint ein, ein statisches Analysewerkzeug im unteren Preissegment. Wir erhielten mehr als 2000 Warnungen. Etwas mehr als eine Woche habe ich gebraucht, um alle Warnungen durchzusehen, den Code im Bedarfsfall zu korrigieren oder die Warnungen durch spezielle Kommentarzeilen zu unterdrücken. Es blieben nur 2 Warnungen, die tatsächlich Probleme hätten bereiten können. Zum einen ein Type-Cast, dessen Ergebnis Compiler-abhängig war. Wir hatten nicht vor, die Plattform zu ändern, also war das nicht relevant für uns. Das zweite Problem war ein nicht beachteter Rückgabewert der Schreibroutine eines EEPROM-Treibers. Hätte die Routine einen defekten Block erkannt, so hätte sie einen Fehlercode zurückgegeben und es wäre ein nochmaliger Aufruf notwendig gewesen. Dieser nochmalige Aufruf unterblieb aber im aufrufenden Modul, weil der Return-Wert nie geprüft wurde.

4.3 Techniken von Analysewerkzeugen im oberen Preissegment

In diesem Unterkapitel wird beschrieben, welche Fähigkeiten statische Analysewerkzeuge im oberen Preissegment anbieten können. Eine sehr gelungene Gegenüberstellung von drei Werkzeugen ist in [Emanuelsson 08] zu finden. Die in diesem Aufsatz genannten Fähigkeiten der Produkte werden hier komprimiert vorgestellt, mit den Eigenschaften eines vierten Produkts ergänzt und die Beschreibung um Produktfeatures aus dem Jahr 2012 erweitert.

4.3.1 Größerer Komfort für den Benutzer

Wenn man ein preisgünstiges Werkzeug zur statischen Code-Analyse erstmals für Code verwendet, so wird das Werkzeug typischerweise eine sehr große Anzahl von Warnungen ausgeben. Der größte Teil der Warnungen wird auf syntaktische Eigenheiten des Codes zurückzuführen sein. Gehorcht die analysierte Software einem Coding-Standard, so kann das Werkzeug im Regelfall so parametrisiert werden, dass es die Eigenheiten des Coding-Standards berücksichtigt und sich die Anzahl der gemeldeten Warnungen damit drastisch reduziert. Gehorcht der Code keinem einheitlichen Coding-Standard, so bedeutet es oft sehr viel Mühe, alle Warnungen zu prüfen und die mehrheitlichen Falsch-Alarme von den wenigen echten Fehlern zu unterscheiden. Es ist den Anwendern von statischer Code-Analyse daher ganz dringend zu empfehlen, von Anbeginn des Entwicklungsprojekts ein Analysewerkzeug einzusetzen und dieses so oft zu verwenden, wie den Compiler. Ein solches Werkzeug ist dann sinnvoll eingesetzt, wenn es keine Fehler meldet, bevor der Code irgendeinem weiteren Verifikationsschritt (Code-Review, Unit-Test …) unterzogen wird.

Wird in einem Projekt automatische statische Code-Analyse mit günstigen Werkzeugen zu spät eingeführt und existiert also schon eine Menge Code, der im schlimmsten Fall keinem Coding-Standard (keinen Programmier-Richtlinien) gehorcht, so sind tausende Falschwarnungen keine Seltenheit.

Ein Analysewerkzeug einzusetzen und seine Warnungen zu ignorieren, macht keinen Sinn. Die Durchsicht von vielen Meldungen (und vielen Falschwarnungen) kostet aber viel Zeit und ist keine besonders spannende Aufgabe. In solchen Situationen ist die Gefahr daher real, dass die Akzeptanz des Entwicklerteams für den Einsatz eines Code-Analysewerkzeugs gering ausfällt und die Einführung von automatischer statischer Code-Analyse nicht an technischer Umsetzbarkeit, sondern an »weichen Faktoren« scheitert.

Werkzeuge im oberen Preissegment haben daher Filter eingebaut, die aus den vielen Meldungen des Basis-Algorithmus die heraussuchen, die tatsächlich relevant sein dürften. Dieses Expertensystem kann natürlich nicht garantieren, dass dabei nicht auch echte Fehler weggefiltert werden. Durch die Filterung ist aber bei einem verspäteten Projekteinsatz mit deutlich weniger Akzeptanzproblemen beim Team zu rechnen als bei kostengünstigen Werkzeugen. Im Idealfall kann man die Filter konfigurieren.

Was ebenfalls zur verbesserten Akzeptanz beitragen kann, ist das Speichern von »bekannten Warnungen« in einer Datenbank und die Möglichkeit, diese wegzufiltern. Wenn alter Code-Bestand übernommen wird und dieser nicht verändert werden darf, ist es auf diese Art sehr einfach möglich, die Analyse nur auf neue Code-Teile zu beschränken.

Auch bieten hochpreisige Werkzeuge zum Teil eine Integration mit Konfigurationsmanagement-Werkzeugen an. Das Analysewerkzeug sagt zu jedem gefundenen Fehler in welchen Versionen der Software er auftritt. Diese Eigenschaft ist

für einen (zu) späten Einsatz des Analysewerkzeugs im Projekt komfortabel. In Projekten mit diszipliniertem Einsatz von automatischer statischer Analyse sollte es nie so weit kommen, dass ein automatisch erkennbarer Fehler überhaupt versioniert wird (also in das Konfigurationsmanagement-Werkzeug eingecheckt wird).

4.3.2 Concurrency Checks

Auch die Palette an durchgeführten Analysen vergrößert sich bei hochpreisigen Werkzeugen. Zum Beispiel wird untersucht, ob bei nebenläufigem (*multi-threaded*) Code doppelte Locks existieren oder Lock-Releases fehlen. Ebenso wird geprüft, ob bei multiplem Locking die Reihenfolge der Lock-Releases umgekehrt zu den Lock-Obtains ist. Durch diese Checks können einfache Dead-Locks abgefangen werden. Mehr zu Locks und Deadlocks in Kapitel 10 und in Kapitel 11.

Auch ist es mit statischer Analyse möglich, mit einer gewissen Treffsicherheit Data Races vorherzusagen. Dass das Erkennen von Data Races für den Benutzer bequem und zuverlässig funktioniert, ist auch im Interesse von Herstellern von Multi-Core-CPUs. Daher sind hier sehr komfortable Werkzeuge am Markt, die völlig gratis sind. Das passt nicht ganz zum Titel dieses Unterkapitels, »Techniken von Analysewerkzeugen im oberen Preissegment«. Wir beschränken uns daher hier auf die Erwähnung der Möglichkeit so einer Analyse und auf einen Querverweis auf Kapitel 10, das sich ausführlich mit Data Races beschäftigt.

4.3.3 Stack-Analyse und erweiterte Kontrollflussanalyse

Mit kleinen Einschränkungen kann durch Analyse des Quellcodes auch festgestellt werden, ob die Größe des Runtime-Stacks ausreicht. So eine Einschränkung ist zum Beispiel die Abwesenheit von Rekursionen. 100 % korrekte Ergebnisse so einer Analyse kann man aber nur erwarten, wenn die Analyse nicht auf Basis des Quellcodes, sondern auf Basis des Object-Codes erfolgt. Nur dann berücksichtigt die Analyse die Eigenheiten und den Optimierungsgrad des verwendeten Compilers. Werkzeuge, die Object-Code analysieren sind manchmal proprietäre Software. So zum Beispiel für den Steuerungsrechner der Ariane-5-Rakete. Es gibt aber auch kommerzielle Werkzeuge, die den Object-Code einer ganze Reihe von Prozessoren unterstützen, zum Beispiel [URL: StackAnalyzer], [URL: GNATstack] oder [URL: IAR].

Da es möglich ist, einen Baum aller Programmzustände zu erzeugen und zu untersuchen, ist es auch möglich, ein Set aller möglichen Kontrollflüsse durch ein Programm zu erzeugen. In dieser riesigen Pfadsammlung lässt sich zum Beispiel für C++-Code durch ein Werkzeug feststellen, ob es für jede Ausnahmesituation (*Exception*) auch eine Behandlung gibt. Ebenfalls gibt es Werkzeuge, die auf diese Art mit ziemlich hoher Treffsicherheit Memory Leaks detektieren.

4.3.4 Erschöpfende Analyse des Zustandsbaums

Wenn der Baum aller möglichen Programmzustände detailliert genug ist, dann lassen sich durch Traversion dieses Baums auch Laufzeitfehler erkennen und – schöner noch – es lässt sich de facto der *Beweis der Abwesenheit von bestimmten Fehlern* erbringen. Das Analyse-Tool durchläuft also das Laufzeitmodell des zu untersuchenden Programms bis in alle Blätter des Baums (untersucht also alle erreichbaren Programmzustände) und kann dann verlautbaren, dass in diesem Baum der Fehlerzustand soundso nicht vorkommt. Die Motivation, solch ein Werkzeug auf den Markt zu bringen, kommt vom missglückten Jungfernflug der Rakete Ariane 5. Dort hatte ein Überlauf bei der Konvertierung eines Gleitkommawerts in einen Integer-Wert mehrere hundert Millionen Euro gekostet. Kein Wunder, dass Werkzeuge zur erschöpfenden Analyse des Zustandsbaums also besonders Überläufe prüfen. Unter anderem wird in dem Baum in solchen Werkzeugen nach folgenden Fehlerzuständen gesucht:

- Overflows/Underflows bei skalaren Variablen
- Overflows/Underflows bei Gleitkommawerten
- Division durch Null
- Array Index ist größer als erlaubt
- Dereferenzierung von Nullpointern

Im Unterschied zur auf Seite 61 in Absatz 4.2.4 vorgestellten Datenflussanalyse von preiswerten Werkzeugen wird hier nicht mithilfe von Heuristiken geschlossen, dass ein Fehlertyp vorhanden ist oder nicht. Wenn das Werkzeug die Abwesenheit einer Nullpointer-Dereferenzierung meldet, dann ist *tatsächlich* keine *zur Laufzeit* im Code.

Natürlich ist so eine erschöpfende Analyse des Zustandsbaums sehr aufwändig. Es sollte auch nicht überraschen, dass Werkzeuge, die diese Analyse anbieten, meist nicht gerade billig sind.

4.4 Statische Security-Analyse (SSA)

Es gibt Werkzeuge für die statische Analyse, die speziell nach Sicherheitslücken im Code suchen. Dabei ist Sicherheit im Sinne von Sicherheit gegen Angriffe von Hackern gemeint (Security). Hacker benutzen oft Programmabstürze und Pufferüberläufe für Angriffe. Die einschlägigen Werkzeuge untersuchen den Quellcode daher nach Programmfehlern, so wie im Buch bisher beschrieben, und zusätzlich analysieren sie die Verwendung von Befehlen und Design-Konstrukten, die von Hackern ausgenutzt werden könnten. In der Programmiersprache C melden diese Werkzeuge zum Beispiel eine gefährliche Verwendung von `gets` und `strcpy`.

In C ist die Software bei einer fehlenden Prüfung auf eine Null-Terminierung von Strings nicht gegen String-Overflows geschützt und damit verwundbar.

Abbildung 4–1 zeigt ein Beispiel. Das Werkzeug meldet eine potenzielle Division durch Null ebenso wie einen möglichen Pufferüberlauf.

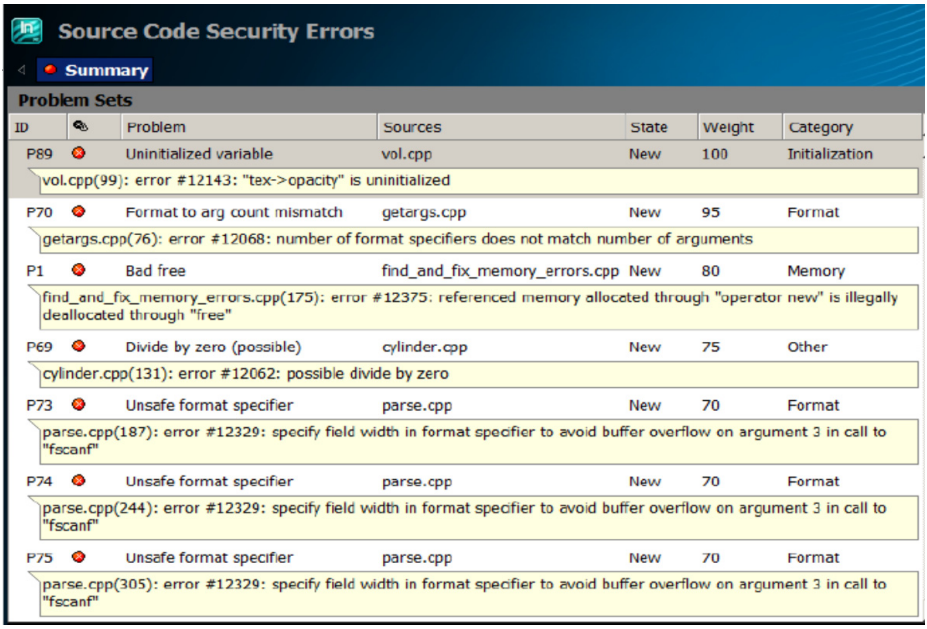

Abb. 4–1 *Ausschnitt eines Screenshots von Intel Inspector XE 2011*

In der letzten Revision des Coding-Standards MISRA-C wurden mehrere Regeln definiert, die es Hackern erschweren sollen, Schadsoftware in Steuergeräte mit Bus-Schnittstellen einzuschleusen. Die Motivation für die Definition dieser neuen Regeln war die berechtigte Sorge, dass Hacker die Personensicherheit gefährden können, wenn sie Steuergeräte in Fahrzeugen angreifen. Der Einsatz eines Werkzeugs, das die Einhaltung dieser Regeln automatisch prüft, ist daher auch ein gewisser Security-Check.

Damit schließt dieses Kapitel die Ausführungen zum Thema automatisches Finden von potenziellen Bugs und potenziellen Verwundbarkeiten und wendet sich einer automatischen Analyse zu, die verwendet wird, um die »innere Qualität« von Programmen zu bewerten.

4.5 Code-Metriken

Der ISTQB definiert *Metrik* als eine Mess-Skala plus das genutzte Messverfahren [ISTQB-D]. Das Wort Metrik entstammt dem Griechischen und bedeutet Kunst des Messens. Unter dem Begriff Code-Metrik versteht man dennoch nicht die Kunst, Quellcode zu vermessen, sondern die Maßzahl selbst. Die Motivation, diese Maßzahlen (werkzeugunterstützt) zu erheben, ist, in kurzer Zeit Eigenschaften des recht abstrakten Objekts Software quantifizieren zu können.

Eine denkbare einfache Software-Metrik ist die Zeilenanzahl einer Funktion oder einer Datei (*Lines of Code*, LOC).[2] Programme werden dank Metriken vergleichbar, zumindest dann, wenn sie in ähnlichen Programmiersprachen erstellt wurden. Aber, wie Peter Liggesmeyer in [Liggesmeyer 09] korrekterweise warnt, ist eine Aussage nur sinnvoll, wenn man mehrere Metriken gleichzeitig betrachtet. Der Blick auf eine Metrik alleine ist irreführend. So kann ein hochoptimierter FFT-Algorithmus mit 500 LOC um Kategorien aufwändiger zu schreiben und zu testen sein als 2000 LOC einer simplen Applikation.

Code-Metriken geben einen Hinweis auf die sogenannte *innere Qualität* von Software. Also die Wartbarkeit, Erweiterbarkeit, Analysierbarkeit, Testbarkeit von Code. Ohne Zweifel kann Code horrend schlecht programmiert sein, also von schlechter innerer Qualität und trotzdem alle Software-Anforderungen erfüllen. Also von guter *äußerer Qualität* sein. Es gibt aber zahlreiche Studien, die den Schluss zulassen, dass innere und äußere Qualität positiv korrelieren. Es ist daher nicht unüblich, dass Käufer von Software mit hohen Integritätsansprüchen den Nachweis des Erreichens von bestimmten Metrik-Schwellwerten einfordern (oder beim Kauf von Quellcode als ersten Schritt Code-Metriken erfassen und sie gegen Schwellwerte vergleichen). Ein solcher Vorschlag für Schranken von Metriken wird in Tabelle 4–1 vorgestellt. Die Metriken sind ein Subset der in [Kuder 08] ausgewählten Metriken. Die Schranken (beziehungsweise zur Akzeptanz vorgeschlagene Wertebereiche) stammen zum Teil ebenfalls aus [Kuder 08], zum Teil aus [ISO 26262] und zum Teil aus den Qualitätsanforderungen an die Software der GMES-Erdbeobachtungssatelliten der ESA. Die Metriken sind um wertvolle Kommentare ergänzt.

2. Einfach dann zumindest, wenn man sich darauf geeinigt hat, welche Kommentarzeilen man mitzählt oder nicht. Verschiedene Tools liefern gerade bei dieser einfachen Metrik verschiedene Ergebnisse.

Code-Metrik	Bereich	Kommentar
Anzahl der Exit-Points	1	Jede Funktion sollte genau einen Austrittspunkt haben. Bei Sprachen ohne Exception Handling kann man die Forderung auch aufweichen, um den Code einfacher zu halten: Zur Fehlerbehandlung sind mehrere Austrittspunkte gestattet, im fehlerfreien Fall gibt es nur ein einziges Function Return.
Anzahl der Sprunganweisungen	0	Die Verwendung von Goto erhöht die Anzahl der möglichen Pfade unnötig und reduziert die Testbarkeit und Wartbarkeit von Code.
Anzahl rekursiver Funktionen	0	Rekursionen haben in Software mit hohem Integritätsanspruch nichts verloren.
Anzahl der Funktionsparameter	0 bis 5	Eine hohe Anzahl von Funktionsparametern birgt ein höheres Risiko eines Integrationsproblems und verursacht großen Verwaltungsaufwand beim Aufruf der Funktion.
Verschachtelungstiefe	0 bis 4	Dies ist üblicherweise die Anzahl der Einrückungen bei Schleifen, if-Statements usw. pro Funktion. Je verschachtelter, desto fehleranfälliger.
Zyklomatische Komplexität von Funktionen	1 bis 12	Die zyklomatische Komplexität einer Funktion ist die Anzahl der Entscheidungen (Verzweigungen) plus eins. Sie ist ein Maß für den Testaufwand beim Basis Path Testing, siehe Abschnitt 6.7. Bei Funktionen mit großen switch/case-Anweisungen (zum Beispiel Nachrichten-Interpretern) ist es üblich, auch größere Werte zu akzeptieren.
Pfadanzahl pro Funktion	1 bis 80	Anzahl der nicht-zyklischen Ausführungspfade durch eine Funktion.
Anzahl dynamischer Objekte oder Variablen	0	Die [ISO 26262] gesteht ein, dass diese Forderung beim Einsatz von modellbasierter Entwicklung nicht erfüllbar sein kann.
Kommentardichte	> 0,2	Die Kommentardichte einer Funktion ist die Anzahl der Kommentarzeilen einer Funktion, dividiert durch die Anzahl der Codezeilen der Funktion.

Tab. 4–1 *Beispiele für elementare Code-Metriken*

Die Motivation für die angegebenen Schranken ist immer die Wahrscheinlichkeit für besser wartbaren beziehungsweise besser testbaren Code. Selbstverständlich kann Code, dessen Metriken alle vorgestellten Schwellwerte verletzen, fehlerfrei sein. Ebenso kann Code, dessen Metriken innerhalb aller angegebenen Schranken liegen, trotzdem schlecht zu testen und schlecht zu warten sein.

Erfahrungsbericht mit Code-Metriken

Ich habe in einem Unternehmen als Entwickler gearbeitet, das Telefonanlagen entwickelte. Das österreichische Tochterunternehmen lieferte die Firmware im C++-Quellcode an das deutsche Stammhaus, wo die Elektronik gefertigt wurde. Die erste (und vermutlich einzige) »qualitätssichernde Maßnahme«, die man im Stammhaus mit dem Firmware-Quellcode unternahm, war zu erheben, wie groß die Kommentardichte im Code war. Die österreichischen Programmierer waren alle lang gediente Mitarbeiter, die sich im Quellcode sehr gut auskannten. Sie hatten wenig Interesse daran, dass ihre Software auch von deutschen Kollegen gut verstanden wurde und der Entwicklungsstandort somit relativ leicht hätte verändert werden können. So fand ich im Code zahlreiche Kommentare der folgenden Form:

```
i = i + 1;  /* erhöhe i um eins */
```

Diese Kommentare ließen den Kommentardichte-Checker grünes Licht geben, haben aber natürlich nichts zur Wartbarkeit und Qualität des Codes beigetragen.

4.6 Werkzeuge für die Automatische Code-Analyse

Die Ausführungen in Abschnitt 4.2 basieren auf den Eigenschaften der Werkzeuge splint und PC-Lint. Vergleichbar damit ist das etwas mächtigere und modernere Werkzeug C-STAT. In Abschnitt 4.3 floss die Übermenge der Features von Klocwork-Produkten, GammaTech CodeSonar, QA-C, der Coverity Development Testing Platform, Polyspace Verifier und Astrée ein. Die letzten beiden Werkzeuge sind, wie die Namen vermuten lassen, Werkzeuge zur Exploration des Zustands[r|b]aums der Software. Alle diese Werkzeuge unterstützen zumindest die Analyse von Quellcode in C/C++.

Es gibt eine ganze Menge anderer Sprachen, für die automatische Code-Analysewerkzeuge am Markt sind. Unter anderem für Ada, Java, C#, Visual Basic, Python und sogar für IEC 61131-3 [Prähofer 12]. Erwähnenswert ist vielleicht, dass es für Java Werkzeuge gratis gibt, deren Funktionalität mit Werkzeugen im mittleren bis hohen Preissegment für C/C++ vergleichbar ist, zum Beispiel das von der NASA bereitgestellte Werkzeug »Pathfinder«.

Für manche Computersysteme ist das Thema Security, also Sicherheit vor unerlaubter und/oder böswilliger Benutzung von großer Wichtigkeit. In diesem Buch wird Security-Tests aber verhältnismäßig wenig Aufmerksamkeit gewidmet. Das bedeutet aber nicht, dass Security nicht auch im Embedded-Bereich wichtig sein kann. Alles ist Angriffsziel von Hackern und Verrückten. Wegfahrsperren für Kraftfahrzeuge werden ebenso geknackt wie die Service-Schnittstelle eines Herzschrittmachers [Halperin 08].

Werkzeuge für statische Security-Checks sind manchmal in Compilern integriert, siehe zum Beispiel Microsoft Visual Studio. An Stand-alone-Werkzeugen

wären zum Beispiel splint für C (Freeware), Fortify SPA für C, C++, C#, Java u.a., Code Assure für C, C++ und Java zu erwähnen.

Werkzeuge zur Erhebung von Code-Metriken machen im einfachsten Fall eine Auflistung von erhobenen Metriken, wie das Komplexitätsmesswerkzeug Testwell CMT++, können aber auch mit anderen Werkzeugen kombiniert sein. So erhebt zum Beispiel QA-C nicht den Anspruch, den gründlichsten Bugfinder am Markt zu haben, dafür vereint das Werkzeug eine grafische Benutzeroberfläche, Source-Code-Navigation, einen Metriken-Checker und eine passable statische Analyse zum Aufspüren von Bugs. Es versteht sich als ein Werkzeug zur Vorbereitung einer manuellen Code-Review. Die erhobenen Metriken und die grafische Aufbereitung derselben sollen helfen, bei der Review den Blick auf das Wesentliche zu konzentrieren. Auch mit freien Versionen von Microsofts Visual Studio können übrigens einige Code-Metriken berechnet werden.

Auch eine Integration von Metriken-Werkzeugen mit Unit-Test-Werkzeugen gibt es. So kann man in Cantata nicht nur Schranken für dynamische Tests definieren, sondern auch für Code-Metriken. Es ist also zum Beispiel definierbar, dass keine 100 % Unit-Tests-Finalisierung angezeigt wird, wenn die zyklomatische Komplexität von C-Funktionen einen definierbaren Schwellwert überschreitet.

Werkzeuge, die eine sehr große Anzahl von einfachen Code-Metriken erheben, diese zu komplexen Metriken verknüpfen und dann (heuristische) Aussagen über Wartbarkeit oder Testbarkeit erlauben, sind zum Beispiel McCabe IQ oder Logiscope. Einen kleinen Ausschnitt aus einem Analysebericht so eines Werkzeugs für Software sehr guter Qualität zeigt Abbildung 4–2.

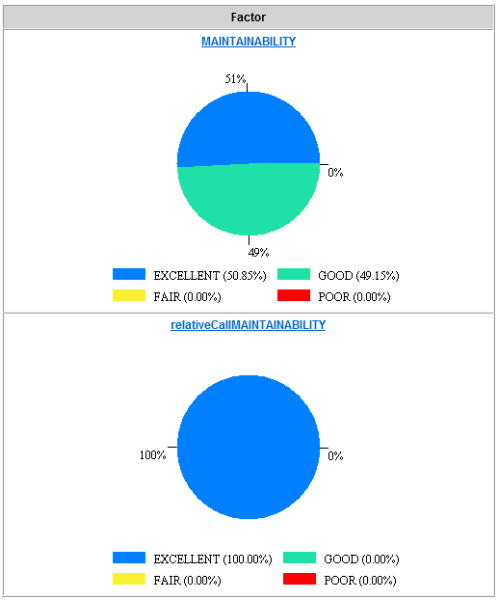

Abb. 4–2 *Ausschnitt aus einem Bericht in HTML-Form von Logiscope*

4.7 Diskussion

Wenn man vergleicht, wie schnell ein potenzieller Bug durch ein statisches Analysewerkzeug, wie in den Abschnitten 4.2 und 4.3 vorgestellt, gefunden wird, und wie lange man benötigt, solche Bugs durch Debugging-Sessions auf Systemebene zu identifizieren, dann ist man schnell davon überzeugt, solche Werkzeuge einzusetzen. Der Werkzeugeinsatz amortisiert sich relativ früh und ist bei Systemen mit Sicherheitsrelevanz nicht wegzudiskutieren. Je später der Einsatz im fortschreitenden Projekt erfolgt, desto mehr sinnlose Warnungen wird man aber beim Einsatz eines günstigen Werkzeugs prüfen müssen. Auch die Wartbarkeit von Code verbessert sich, wenn man gleich von Anbeginn des Projekts automatische statische Analyse einsetzt und danach trachtet, immer null Warnungen zu erhalten: Viele gefährliche Code-Konstrukte werden von vornherein vermieden und eine Analyse einer eventuell als gefährlich eingestuften Code-Passage zu einem späteren Zeitpunkt der Wartung kann unterbleiben.

Die Vielzahl unterschiedlicher Fehlertypen, die ein Low-Cost-Werkzeug findet, kann zwar einen kräftigen Dämpfer für einen späten Einsatz bedeuten, es können aber auch Fehler gemeldet werden, die von den Filtermechanismen der hochpreisigen Werkzeuge fälschlicherweise ausgeblendet/wegpriorisiert werden. In einer Studie [Zheng 06] wurde beschrieben, dass ein Low-Cost-Werkzeug etwa doppelt so viele Fehler wie ein hochpreisiges Produkt erkannte – sowohl hinsichtlich der Fehleranzahl wie auch in Bezug auf die Verschiedenartigkeit der Fehlertypen. Ein anderer Bericht bestätigt diese Studie. Wie groß die Schere der Anzahl der Fehlermeldungen in großen Projekten werden kann, berichtet [Emanuelsson 08]: 1,2 Millionen Defekte mit einem günstigen Tool und nur 40 Fehlermeldungen mit einem hochpreisigen Werkzeug.

Auch beim Einsatz von Werkzeugen der Art von Astrée und Polyspace muss man mit so manchem Bedienungsaufwand rechnen, wenn man den Code zum Beispiel nach potenziellen Überläufen und ähnlichen Laufzeitfehlern absucht. Der Lohn für diese Arbeit ist der Nachweis der Abwesenheit von bestimmten Fehlern. Einen Nachweis der Abwesenheit von Fehlertypen können Bugfinder-Werkzeuge nicht liefern.

Auch der gewissenhafte Einsatz von Werkzeugen kann aber eine Code-Review durch einen Menschen nicht ersetzen. Nur ein menschlicher Review-Partner kann unangepasstes Design oder die fehlerhafte Umsetzung von Anforderungen erkennen. Die Analyse des Codes durch einen Kollegen und die automatische Analyse durch ein Werkzeug sind daher großteils komplementäre Vorgänge zur Qualitätssicherung. Beide Verifikationsschritte sollten also eingesetzt werden.

4.8 Fragen und Übungsaufgaben

Frage 4.1: Die Kommentare im folgenden Listing schlagen sieben Warnungen vor. Welche dieser sieben Warnungen würde ein aggressives Werkzeug zur statischen Analyse tatsächlich melden? Welche nicht? Warum?

```
/* C-Programm */
int Hoo();                    /* 1, incomplete prototype */
char cUartData = (char) 0xFA;

main()
{
   unsigned uiA = 12;           /* 2, uiA is never used */
   int i, a[10] = {0,0,0,0,0,0,0,0,0,0};
                              /* 3, wrong number of init:s */

   for (i = 0; i <= 10; i++)
   a[i] += i;             /* 4, index exceeds buffer size */
   for (i = 0; i <= 3; i++); /* 5, suspicious semicolon */
   {
      a[i] = (int) Hoo;              /* 6, strange cast */
      a[i] -= cUartData;
      a[i] += Hoo();
   }
}                           /* 7, no return value of main */

int Hoo(int x)
{
   return x + 2;
}
```

Frage 4.2: Warum können Sie nicht mit einer Quellcode-Analyse eines Off-the-Shelf-Werkzeugs zuverlässig den Stack-Verbrauch bestimmen?

Frage 4.3: Können Sie mithilfe von Code-Metriken die Qualität einer Software-Bibliothek beurteilen, die Sie als Object-Datei (also nicht im Quellcode) erhalten haben?

Frage 4.4: Können Sie mithilfe von Code-Metriken eine hinreichend zuverlässige Aussage über die Qualität von Quellcode machen?