Static Analysis of Embedded Systems

Proseminar: Technische Informatik

Betreuer: Prof. Dr. Marcel Kyas

Konrad Reiche

15. Januar 2010

Inhaltsverzeichnis

1 Einleitung

Bei einem Desktop- bzw. Workstation-PC sind Programmabstürze Ursache von fehlerhafter Programmierung. Bei umfangreicher Software ist dies nicht vermeidbar. Der Schaden ist im Anwendungsbereich beträchtlich. Obwohl die Daten meist durch Backups redundant vorliegen, entsteht ein deutlicher Zeitaufwand durch Neustart des Programms bzw. Computers und der Aufarbeitung des Backups zum aktuellen Datenbestand. Das Problem Programmabsturz ist bei eingebetteten Systemen[1] sogar von noch größerer Dimension.

Eingebettete Systeme zeichnen sich durch Anwendungen aus, die in Echtzeit und mit maximaler Zuverlässigkeit arbeiten. Weiterhin werden eingebettete Systeme durch Hardware realisiert. Im Konsumbereich werden diese Systeme in hoher Stückzahl produziert. Durch den Verkauf an eine Vielzahl von Kunden verteilen sich diese und sind außerhalb der Reichweite der Entwickler. Dadurch hat es eine große Bedeutung, dass die verwendete Software nicht mehr nachträglich verändert werden sollte. Die Software muss bereits bei Auslieferung korrekt funktionieren und sollte keine Fehler aufweisen. Kommt es beispielswiese zum Versagen eines Herzschrittmachers wäre der Schaden, verursacht durch Software-Fehler, lebensgefährlich.

Um solch undefiniertes Verhalten zu vermeiden wird die Software auf fehlerhaftes Verhalten überprüft. Eine Möglichkeit ist das Hoare-Kalkül. Als Eingabe erhält es den Algorithmus des zu testenden Programms und eine Vor- und Nachbedingung. Vor- und Nachbedingung sind mathematische Beschreibungen der gewünschten Zustände. Das Ergebnis ist eine Antwort auf die Frage: Erfüllt der Algorithmus die Vor- und Nachbedingung? Diese Verifikation ist nicht entscheidbar[2].

Weiterhin gibt es *Model checking*. Die Eingabe ist ein endliches Modell des Programms und eine Spezifikation gegen die das Modell geprüft wird. Entweder erfüllt das Modell die Spezifikation oder es wird ein Gegenbeispiel gefunden. Dieses Verfahren ist entscheidbar.

Ein weiteres Verfahren ist das dynamische Testen. Das Programm wird ausgeführt und mit ausgewählten Eingaben getestet. Entweder wird das erwartete Ergebnis ausgeben oder nicht. Da man nicht alle Eingaben testen kann, ist dieses Verfahren ebenfalls nicht entscheidbar.

Das vierte Verfahren zur Verifikation ist die statische Analyse. Hier ist die Eingabe lediglich der Quelltext des Programms. Das besondere im Gegensatz zu den bisher genannten Verfahren ist, dass keine Spezifikation benötigt wird. Die statische Analyse ist in der Anwendung eine Zusammenfassung von mehreren kleinen spezialisierten Analysen. Das bedeutet, eine Spezifikation wird gar nicht benötigt. Alle zu testenden Eigenschaften sind in den einzelnen Analysen bereits festgelegt [4]. Das Ergebnis dieser Analysen ist eine Verifikation bzw. Falsifikation der Eigenschaft.

[1] Eingebettete Systeme: Hard- und Softwaresysteme, die in einer technische Umgebung fest eingebunden sind und Informationsverarbeitung sowie Steuerungsprozesse übernehmen. Eingebettete Systeme sind anwendungsspezifisch und müssen mit minimalen Speicher- und Prozessorressourcen auskommen. Diese Systeme kommen u.a. in der Luft- und Raumfahrt, Automobilproduktion, aber auch in der Konsumelektronik wie Handys oder MP3-Player zum Einsatz.

[2] Entscheidbarkeit: Ein Verfahren ist entscheidbar, wenn ein Algorithmus für dieses Verfahren existiert und der Algorithmus auf alle Eingaben der Elemente des Verfahrens ein Ergebnis berechnet und terminiert.

Wenn diese Antwort nicht eindeutig festgelegt werden kann, lässt man diese mit einer Markierung der möglichen Position im Quelltext offen. Auf Grund der letzten genannten Eigenschaft ist die statische Analyse entscheidbar.

Dies wirft gegenüber den anderen drei Verfahren einige Vorteile auf. Alle Verfahren müssen zu mindestens teilautomatisiert werden, da die Software mittlerweile zu umfangreich ist. Das Hoare-Kalkül wird mit Computern durch Theorembeweiser automatisiert. Die Algorithmen sind aber sehr komplex, erreichen exponentielle Laufzeit und sind damit sehr zeitintensiv. Ähnliches gilt für dynamische Testvorgänge. Es können nicht alle Eingaben getestet werden und um diese dennoch möglichst umfangreich zu halten wird ebenfalls viel Zeit benötigt. Der Markt für eingebettete Systeme unterliegt jedoch wie jede andere wirtschaftliche Produktion einem hohen Zeitdruck für die Fertigung und dem Verkauf. *Model checking* ist auch nur begrenzt einsetzbar, da die Endlichkeit des Modells eine Restriktion darstellt. Möchte man ein Modell für die Eingabe einer unendliche Menge von Elementen konstruieren, so ist dies nicht mehr möglich.

Die statische Analyse ist sehr genau in der Überprüfung der Eigenschaften, automatisiert, entscheidbar und kann mittlerweile sogar schon während des Entwicklungsprozesses verwendet werden und nicht erst nach Fertigstellung der Software.

Die Anzahl der überprüfbaren Eigenschaften eines Programms ist sehr groß. Dementsprechend gibt es viele Methoden in der statischen Analyse. Eine einzelne Analyse beschränkt sich beispielsweise auf die Berechnung des möglichen Wertebereichs von Variablen. Eine weitere Analyse bestimmt das Vorzeichen von Zahlen. Durch die Kombination der Vielzahl von Analysen wird die statische Analyse sehr umfangreich in der Anzahl der überprüfbaren Eigenschaften eines Programms. Das Finden von Fehlern die bei Ausführung zu einem undefinierten Zustand führen, einem Laufzeitfehler, ist die herausfordernde Aufgabe von verschiedenen Analysen.

1.1 Stand der Technik

Zu den einfacheren Analysen gehört die Typüberprüfung. Dabei wird die Typverträglichkeit von Zuweisungen überprüft und ob das Durchführen von *type casts* zulässig ist. *Code convention analysis* überprüft den verwendeten Programmierstil, wie Variablennamen. Es sind Eigenschaften die nicht unbedingt zu Laufzeitfehlern führen müssen, dennoch ist es statische Analyse.

Sehr häufige Laufzeitfehler sind *buffer overflow*, bei denen eine große Datenmenge in einen zu kleinen reservierten Speicherbereich geschrieben wird und eventuell der Bereich des Programmspeichers überschrieben wird. Bei *out of bound reads* wird beispielsweise eine Zählvariable über ein Feld iteriert und die Zählvariable wird größer als die Gesamtlänge des Feldes. Bei der *null pointer dereference analysis* wird überprüft, ob ein Zeiger dereferenziert wird der auf `null` zeigen könnte. Speicherlecks sind ein Problem, bei dem auf Zeiger zugegriffen wird, die auf keine Adresse im Speicher mehr zeigen.

Die statische Analyse wird durch Werkzeuge durchgeführt, die solche Probleme aufdecken sollen.

Auf Grund der Verwendung für eingebettete Systeme haben diese Werkzeuge zunehmend Bedeutung in der Industrie gewonnen. Zu den wichtigsten gehören PolySpace, Global C Surveyor und Astrée, die in dieser Arbeit betrachtet werden. Sie sind nicht nur in der Lage die Existenz von den genannten Fehlern zu zeigen, sondern teilweise auch die Abwesenheit von allen Laufzeitfehlern.

Die statische Analyse wird jedoch nicht nur zum Finden von Fehlern verwendet. Ein Beispiel ist die WCET-Analyse durch das Werkzeug aiT von AbsInt [12]. Es ist eine Approximation der oberen Schranke (*worst case execution time*), die ein Programm für seine Ausführungszeit benötigt.

Schlussendlich ist die statische Analyse auch nicht uneingeschränkt anwendbar. Das Halteproblem aus der theoretischen Informatik macht deutlich, dass es nicht entscheidbar ist, ob ein Programm in jedem Fall terminieren wird [2]. Um dies dennoch zu realisieren und die statische Analyse entscheidbar zu machen, wie oben angegeben wurde, werden unterschiedliche Verfahren verwendet. Viele der oben genannten Fehler sind abhängig von den Werten die Variablen annehmen können. Wie sich diese während der Programmausführung verändern wird durch die beiden folgenden Verfahren erläutert.

2 Control Flow Analysis und Data Flow Analysis

2.1 Control Flow Analysis

Die Kontrollflussanalyse ist die Bestimmung von Kontrollstrukturen in einem Programm zur Abstraktion eines Programms in einzelne Programmfragmente. Kontrollstrukturen sind Anweisungen im Code, die den unterschiedlichen Verlauf des Programms steuern. Insbesondere für funktionale Programmiersprachen und Assembler ist es sinnvoll, da im Gegensatz zu imperativen Sprachen die Kontrollstrukturen nicht als Bestandteil der Syntax vorliegen.

Dennoch verbleibt die Kontrollflussanalyse auch bei imperativen Programmiersprachen von Interesse. Gründe dafür sind, dass viele Sprachen GOTO-Anweisungen verwenden oder die Analyse des Maschinencodes, in dem die Kontrollstrukturen nicht mehr direkt erkennbar sind.

Die Analyse wird mit Hilfe der Konstruktion eines *control flow graphs* (CFG) durchgeführt. Einzelne Knoten repräsentieren Basisblöcke. Basisblöcke sind aufeinander folgende Anweisungen, die nur am Anfang oder am Ende betreten bzw. verlassen werden können. Die Kanten repräsentieren Kontrollübergänge (Sprunganweisungen) des Programms. Zusammen bilden sie den Kontrollfluss des Programms, bestehend aus eventuellen Schleifen, GOTO-Anweisungen, bedingten Anweisungen, etc [9].

In der Programmverifikation ist der CFG von großer Bedeutung, da durch ihn ein Großteil der Datenflussanalyse erst ermöglicht wird. Eine zentrale Aufgabenstellung der Kontrollflussanalyse ist das Aufdecken von Schleifen. Die wichtigsten Informationen sind Inhalt der Schleifenköpfe und an welchen Stellen sich der Kontrollfluss teilt und zusammenfließt.

Insbesondere haben `while`-Schleifen eine gewichtete Bedeutung für die Verifikation von Programmen. Für Programme mit `while`-Schleife lässt sich nicht immer ein Entscheidungsverfahren finden. Im Hoare-Kalkül muss dafür eine Invariante[2] und eine Terminationsfunktion[3] gefunden werden. In der statischen Analyse wird die Schleife approximiert. Dieses Konzept der Näherung wird in Kapitel 3 ausführlich erläutert.

Ist das Programm durch einen CFG repräsentiert, können weitere Analyse auf diesem Modell durchgeführt werden. Zu diesen Analysen gehört die Datenflussanalyse [9].

2.2 Data Flow Analysis

Die Datenflussanalyse ist ein Verfahren zur Ermittlung von möglichen Werten bzw. Wertebereichen des vorliegenden Programms. Für die Bestimmung dieser wird der eben genannte CFG benötigt. Im Gegensatz zur Kontrollflussanalyse steht aber der Inhalt der Knoten im Zentrum: Zuweisungen und Operationen [5].

Dabei wird die Zuweisung von Werten an Variablen und die Verwendung dieser im Kontext des Programms festgehalten. Die Wichtigkeit der Datenflussanalyse spiegelt sich in dem Aufdecken von häufigen Programmfehlern wieder. Möchte man auf Division durch 0 testen, so muss für die Anweisung x/y überprüft werden, welche Werte y annehmen kann.

Zu den wichtigsten Analysen der Datenflussanalyse gehört die *pointer analysis*. Es wurden mittlerweile viele Verfahren entwickelt. Die beiden bekanntesten sind *alias analysis* und *points-to-analysis*. Zeigen zwei oder mehr Zeiger auf dieselbe veränderliche Adresse, so handelt es sich um ein *alias* [6]. In der *points-to-analysis* werden alle Verweise ermittelt auf die ein Zeiger zeigen könnte. Folgendes Beispiel soll die Relevanz der *pointer analysis* deutlich machen:

```
1 int *x, *y;
2 *x = 3; // Der Zeiger x verweist auf eine Adresse eines Integer-Wertes
3 *y = 5; // Der Zeiger y verweist auf eine andere Adresse eines Integer-Wertes
4
5 x = y; // Der Zeiger x verweist jetzt auf dieselbe Adresse wie y
6
7 free(y); // Der Speicher auf den y verweist wird freigegeben
8 free(x); // Speicherzugriffsfehler: Der Speicher von y existiert nicht mehr
```

Solche *alias* führen sehr häufig zu Laufzeitfehlern. Wenn es sich dabei sogar um Zeiger auf Zeiger handeln würde, könnte es sehr schnell zum Überschreiben von wichtigen Speicherstellen kommen. Dies könnte auch geschehen ohne das es zu einem Programmabsturz führt. Das Resultat wäre ein falsch berechnetes Ergebnis ohne Hinweis auf die Existenz eines Fehlers.

Die Basiskonzepte für die *pointer analysis* sind wie folgt aufgebaut. Die *flow-sensitive* (*flow-insensitive*) betrachtet (vernachlässigt) den Kontrollfluss des Programms. Es wird vermerkt (nicht vermerkt) zu welchem Zeitpunkt auf eine Adresse verwiesen wurde. In dem Ansatz der

[2]Invariante: Ein mathemathisch beschriebener logischer Ausdruck der Eigenschaften von einem Programm, Algorithmus oder Programmfragment beschreibt. Die Invariante gilt, wenn dieser logische Ausdruck während der gesamten Ausführung wahr ist.

[3]Terminationsfunktion: Eine Funktion, welche die Schleifenbedingung in Abhängigkeit setzt. Das Programm terminiert, wenn die Invariante gilt und die Funktion streng monoton fallend ist.

context-sensitive (*context-insensitive*) Methode wird der Zeigerzugriff im Kontext eines Funktionsaufrufs betrachtet (vernachlässigt). Eine Funktion kann unter verschiedenen Bedingungen aufgerufen werden, zum Beispiel verschiedene Funktionsargumente. Diese werden bei der *context-sensitive* Methode berücksichtigt. Für den *context-insensitive* Ansatz würde die *pointer analysis* Funktionsaufrufe auf einfache GOTO-Anweisungen reduzieren [6].

Solche Konzepte lassen sich nicht immer mit dem konkreten Programm umsetzen. Für die aufgezählten Analysen muss eine Näherung an den eigentlichen Programmkontext durchgeführt werden. Warum und wofür das nützlich ist wird mit dem Prinzip der abstrakten Interpretation im folgenden Kapitel erläutert.

3 Abstract Interpretation

In der Einleitung wurde festgestellt, dass es unmöglich ist, eine Analyse durchzuführen ohne den Quellcode mathematisch zu beschreiben. Patrick Cousot und seine Frau Radhia Cousot haben dafür ein mathematisches Fundament entwickelt: Abstrakte Interpretation. Es ist die Approximation von mathematischen Strukturen zur Beschreibung der Semantik eines Programms [2].

Programme beschreiben Berechnungen über Werte. Diese Beschreibung ist die Semantik eines Programms. Genauer formuliert ist es die konkrete Semantik eines Programms. Die konkrete Semantik ist eine Formulierung von allen möglichen Ausführungspunkten in allen möglichen Ausführungszuständen des Programms [2].

Die konkrete Semantik ist nicht entscheidbar, daher nutzt abstrakte Interpretation die konkrete Semantik für die Konstruktion einer abstrakten Semantik. Die abstrakte Semantik ist die Beschreibung einer Berechnung über einer Menge von abstrakten Werten. Die konkreten Werte eines Programms für arithmetische Berechnungen sind beispielsweise Integer-Werte. Abstraktion bedeutet eine Menge zu wählen, die diese konkrete Menge repräsentiert. Ein Beispiel aus der Arithmetik [2] verdeutlicht dies:

$$-13 \cdot 7 = -91$$

Gegeben sei die abstrakte Menge $Label_A = \{(+), (-), (\pm), (0)\}$. Die abstrakte Multiplikation $\odot$ ist durch die linke Tabelle beschrieben.

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

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

Abbildung 1: abstrakte Multiplikation und Addition

Die *detection of sign analysis*, eine Analyse zur Bestimmung des Vorzeichens von Integer-Werten, würde diese Information folgendermaßen verarbeiten:

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

Durch die Abstraktion wird nicht mehr mit den Werten gerechnet, sondern mit deren Eigenschaften. Das Ziel ist eine Abstraktion zu finden, so dass von den Eigenschaften der abstrakten Semantik auf Eigenschaften der konkreten Semantik geschlossen werden kann. Diese Eigenschaften werden mathematisch beschrieben und heißen *abstract domains*. Die mathematische Grundlage von Cousot für diesen Zusammenhang ist die Verwendung von Galois-Verbindungen.

$$\alpha : K \to A,\ \gamma : A \to K$$

Galois-Verbindungen sind ein paar monotone Funktionen $\langle \alpha, \gamma \rangle$. Die Abstraktionsfunktion α bildet konkrete Werte auf abstrakte Werte ab. Das Gegenstück ist die Konkretisierungsfunktion γ, welche die abstrakten Werte auf alle konkreten Werte mit der Eigenschaft des abstrakten Wertes abbildet. Wird die Reihenfolge erst α dann γ eingehalten wird sichergestellt, dass eine Menge entsteht, die mindestens genauso groß ist wie vor der Abstraktion [2].

Die Abstraktion zu einer festgelegten abstrakten Menge kann jedoch auch zu Ergebnissen führen, die einen geringen Informationsgehalt bereithalten. Mit den Elementen aus $Label_A$ kann mit der abstrakten Addition, durch Abb. 1 (rechte Tabelle) beschrieben, für folgendes Beispiel keine genaue Aussage getroffen werden:

$$-5 + 19 = 14 \Rightarrow -(+) \oplus (+) = (-) \oplus (+) = (\pm)$$

Die abstrakte Interpretation verallgemeinert die Menge der konkreten Werte zu Eigenschaften. In manchen Fälle, wie in diesem Beispiel, ist die Verallgemeinerung jedoch so stark, dass aus dieser Eigenschaft kein Informationsgewinn erzielt werden kann. Die Eigenschaft $(\pm)$ trifft auf alle Integer-Werte zu. In der statischen Analyse wird eher eine sehr allgemeine Aussage getroffen als eine fehlerhafte mit hohem Informationsgehalt.

Dies ist charakteristisch für die statische Analyse, denn als Ergebnis gibt es immer nur zwei Ergebnistypen:

- Ja, die Eigenschaft trifft zu
- Ja/Nein? Vielleicht trifft die Eigenschaft zu, vielleicht auch nicht

In der Einleitung wurde erklärt, von welcher Wichtigkeit die Ergebnisse dieser Analyse sind. Die Ergebnisse müssen korrekt sein. Andernfalls ist nicht sichergestellt, ob der dadurch beschriebene Code-Abschnitt bzw. das Programm korrekt ist.

Um fehlerhaftes Verhalten von Programmen zu erkennen werden Sicherheitseigenschaften spezifiziert. Es sind Aussagen über Eigenschaften des Programms, die durch eine endliche Ausführung

widerlegt werden können. Ein Beispiel wäre: "Der Wert der Variable x wird immer < 5 sein". Weitere Beispiele sind Invarianten oder die "Abwesenheit von Laufzeitfehlern". Zum Beweisen der Sicherheitseigenschaften muss gezeigt werden, dass der Schnitt aus der konkreten Semantik und der Menge die einen fehlerhaften oder undefinierten Zustand beschreibt leer ist.

Die konkrete Semantik kann auf Grund von Unentscheidbarkeit nicht verwendet werden. Bei der Wahl der *abstract domain* kann es jedoch auch zu einer fehlerhaften Abstraktion kommen. So könnten in einer möglichen Abstraktion Zustände nicht betrachtet werden, die aber in der konkreten Semantik enthalten sind. Dieses Problem der Unterapproximation wird durch Verwendung von Galois-Verbindungen ausgeschlossen.

Der gegenteilige Fall die Überapproximation ist aber auch möglich. Es werden Ausführungspunkte in Betracht gezogen die gar nicht existieren. Es handelt sich um *false positive*, falsche Fehlermeldungen, wenn sich diese Ausführungspunkte mit fehlerhaften Zuständen des Programms überschneiden. Um dies zu vermeiden wird die *abstract domain* zu einer konkreteren *abstract domain* verfeinert, die näher an der konkreten Semantik ist. Hierbei sei bemerkt, das statische Analyse sehr komplex bezüglich der Laufzeit werden kann. Eine zu konkrete *abstract domain* kann in einer sehr hohen Laufzeit resultieren. Im Gegenzug kann eine konkrete *abstract domain* sehr genaue Ergebnisse liefern und ist demzufolge in der Lage eventuell mehr Fehler aufzudecken.

Ein weiteres Beispiel sind Intervalle, eine Approximation von Minimum- und Maximum-Werten durch Intervallarithmetik [2]. An diesem Beispiel soll verdeutlicht werden, wie differenziert sich die Analyse durch unterschiedliche Ebenen der Abstraktion gestalten kann.

3.1 Beispiel

Betrachten wir folgendes Programm:

```
1  int x = 3;
2  int y = 5;
3  int z;
4
5  x = 3*x + y;
6
7  while (y < 20)
8  {
9    z = (2*x)/(x-y);
10   y++;
11 }
```

Ein kritische Operation ist die Division durch $(x - y)$, da nicht geklärt ist, ob der Fall $x - y = 0 \equiv x = y$ eintritt. Dies würde zu einer Division durch Null führen, die für Integer-Werte nicht definiert ist. Für eine Analyse sind einzelne Programmzustände nicht von Bedeutung. Stattdessen genügt eine Abstraktion auf das Wertepaar (x, y).

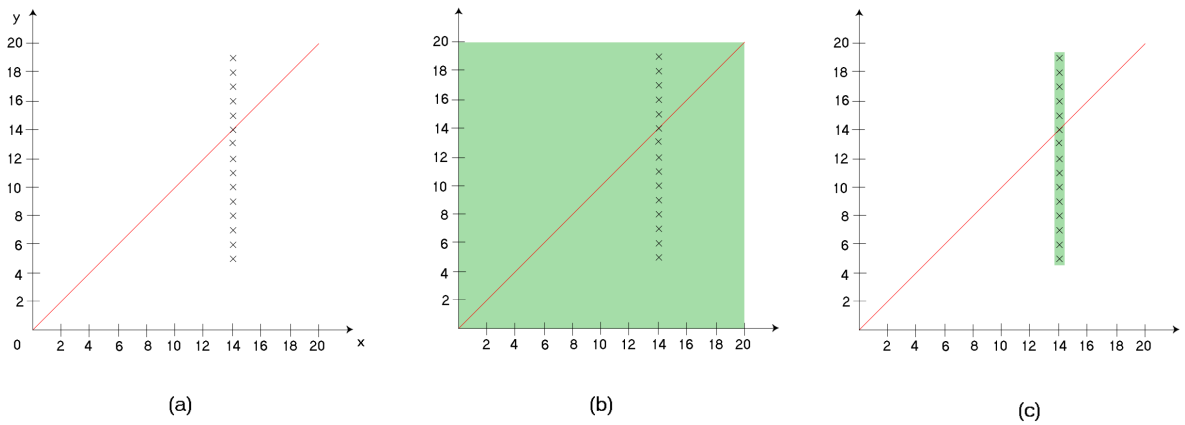

Abbildung 2: Abstraktions-Ebenen

Die konkrete Semantik wird durch Abb. 1(a) dargestellt. Wäre die Möglichkeit der Entscheidbarkeit gegeben, könnte eine Berechnung überprüfen, ob der Graph $f(x) = x$, der den Fall $x = y$ abdeckt, ein Wertepaar schneidet. Da keine Entscheidbarkeit gegeben ist wird abstrahiert. Bei Betrachtung der abstrakten Addition und Multiplikation, wie im Beispiel oben, ergibt sich, dass x und y immer positiv bleiben, Abb. 1(b).

Dies genügt jedoch nicht für das gegeben Problem, die *abstract domain* muss konkretisiert werden. In diesem Fall wird es durch numerische Analyse mit Intervallen realisiert:

`1: int x = 3`	$\Leftrightarrow x \in [3,3]$
`2: int y = 5`	$\Leftrightarrow y \in [5,5]$
`5: x = 3*x + y`	$\Leftrightarrow [3,3] \cdot [3,3] + [5,5] = [3 \cdot 3 + 5, 3 \cdot 3 + 5] = [14,14]$
	$\Leftrightarrow x \in [14,14]$
`7: while(y < 20)`	$\Leftrightarrow$
`10: y++;`	$\Leftrightarrow y \in [5,20]$
`9: z = (2*x)/(x-y)`	$\Leftrightarrow (x-y) : [14,14] - [5,20] = [14-5, 14-20] = [-6,9]$

$$\texttt{(x-y)} \in [-6,9] : 0 \in [-6,9]$$

Aufgrund der Iteration von y durch $y = y + 1$ kann durch Extrapolation das Intervall für die gesamte Schleife berechnet werden. Da sich y in jedem Schritt um 1 erhöht und 0 in dem Intervall enthalten ist, würde es bei Ausführung zu einer Division durch Null kommen. In Abb. 1(c) wird die numerische Analyse durch das Intervall verdeutlicht. Durch die Abstraktion werden Informationen über die Wertemenge der Zahlen gewonnen.

4 Werkzeuge

4.1 PolySpace, C Global Surveyor und Astrée

PolySpace ist ein universales statisches Analyse Tool zur Aufdeckung von Laufzeitfehlern in Ada, C, C++ und Java. Es wurde bereits von der NASA verwendet. Zu den Merkmalen gehören:

- Datei- und Klassen-basierte Prüfung von Softwarekomponenten
- Techniken aus der abstrakten Interpretation

In PolySpace wird der Ansatz verfolgt die Analyse nicht erst auf abgeschlossenen Quellcode-Projekt zu verwenden, sondern bereits während der Entwicklungszeit einzusetzen [11]. Ist eine Funktion, Klasse oder Datei bereits fertig geschrieben, kann der Code analyisiert werden. Heute besteht geschriebene Software aus mehreren tausend Zeilen von Code. Da diese Analysemethoden meist sehr aufwendig sind dauert das Verfahren von mehreren Stunden bis mehreren Tagen [1].

Eine Möglichkeit ist die verwendete Abstraktion, beschrieben in Kapitel 3, grob zu halten. Damit geht jedoch Ungenauigkeit einher. Stattdessen nutzt PolySpace Cloud Computing. Die Software C/C++ PolySpace Server bietet dazu ein Server-Modell an. Dabei schicken PolySpace Clients Teilverifizierungen an die Computer Cluster, die diese weiterverarbeiten.

Da die NASA PolySpace sehr konkret für ihre Raumfahrtsysteme verwendete, stieg das Interesse nach einer Spezialisierung. Die Idee ist es die Analysealgorithmen auf bestimmte Programme zurechtzuschneiden. In der Raumfahrt wären das zum Beispiel Flugkontrollsysteme und digitale Signalverarbeitungen.

Daraus enstand ein von der NASA Ames Research Center entwickelter Analyse Prototype: C Global Surveyor. Im direkten Vergleich lieferte das entwickelte Werkzeug auch beträchtliche Effienzunterschiede [3]:

Analyse Tool	Code	Dauer	Genauigkeit
PolySpace	20-40 KLOC[3]	8-12h	80%
C Global Surveyor	280 KLOC	2-3h	90%

Aus Sicht der Erfolge von statischer Analyse sei noch Astrée zu nennen. Astrée ist ein statische Analyse Werkzeug für eingebettete Systeme entwickelt von Patrick Cousot als Projektleiter. Die Zielsetzung war nicht nur das Aufdecken von Laufzeitfehlern, sondern auch das Überprüfen der Sicherheitsheitseigenschaft "Abwesenheit sämtlicher Laufzeitfehler" [10]. Dieser Unterschied sei hier hervorzuheben. Werkzeuge finden im allgemeinen Fehler. Sollten sie jedoch keine Fehler finden, ist nicht sichergestellt, dass das Programm keine möglichen Laufzeitfehler aufweisen wird.

Auch wenn dies auf Grund der Unentscheidbarkeit nicht immer möglich ist, gelang es Astrée in der Luft- und Raumfahrt die Sicherheitseigenschaft "Abwesenheit von allen Laufzeitfehlern" zu zeigen. Astrée ist damit auch heute noch eine vielgenutzte Anwendung im europäischen Raum.

[3]KLOC (Thousand Lines of Code): KLOC ist die Anzahl der verwendeten Zeilen Code gezählt in tausender Schritten.

Zu den Erfolgen zählte im November 2003 vollautomatisch und automatisiert die Abwesenheit von Laufzeitfehlern im Flugkontrollsystem des Airbus A340 fly-by-wire Systems nachzuweisen. Ein C-Pogramm mit 132000 Zeilen Code, das in 1h 20min analyisiert wurde. Im April 2008 wurde ebenfalls vollständig und automatisch die Abwesenheit von Laufzeitfehlern in der automatischen Dockingsoftware der Jules Vernes Automated Transfer Vehicle verifiziert. Das Jules Vernes Automated Transfer Vehicle ermöglichte es der ESA Versorgungsgüter an die ISS zu liefern [10].

Eine weitere nennenswerte Eigenschaften ist die Nutzung von Parametern. Diese Parameter ermöglichen dem Benutzer eigene Richtlinien einzuführen. Zum Beispiel das Verbieten von modularer Arithmetik auf Integer-Werte. Dies ist vorteilhaft für eingebette Systeme, da es Portierungen und Erweiterungen zulässt.

4.2 aiT - WCET-Analyse

aiT ist ein Werkzeug das ebenfalls statische Analyse automatisiert hat. Im Gegensatz zum bisherigen Umfang dieser Arbeit konzentriert aiT sich aber nicht auf das Finden von Laufzeitfehlern. Es untersucht die oberen Schranken der Ausführungszeit von Systemen: *Worst case execution time* (WCET) [12].

Die Herausforderung ist dabei die Hardwareeigenschaften in die Ausführungsdauer mit einzubeziehen. Während sich die Komplexitätstheorie mit relativer Ausführungszeit außeinandersetzt, ist es vonnöten bei zeitkritischen Anwendungen die tatsächliche Ausführungszeit in Sekunden festzustellen. Zu den kritischten Komponenten gehören dabei Prozessorcache und -pipelines, da die Ausführungszeit von vorrangegangen Programmausführungen abhängt. Diese müssen jedoch mit berücksichtig werden, da ohne sie die Performance um bis zu 97% einbrechen würde [12].

aiT analyisiert dafür den bereits compilierten Code in Form von Objektcode. Aus dem Objektcode werden mittels CFG Schleifen rekonstruiert. Die folgende Datenflussanalyse berechnet Adress- und Wertebereiche. Danach werden mittels Cache- und Pipeline-Analyse mögliche Speicherzugriffe und Verhalten der Prozessor-Pipeline festgehalten. Aus dem enstandenen Kontrollfluss wird schließlich der Ausführungpfad für das Ergebnis ausgewählt, der bei Ausführung am längsten benötigen würde. Dieses Kriterium ist nötig um die Eigenschaft der obersten Schranke für die Ausführungszeit einzuhalten [12].

5 Bedeutung für eingebettete Systeme

Die statische Analyse gewinnt für eingebetteten Systemen zunehmend an Bedeutung und ist mittlerweile bindend, denn statische Analyse erweist sich bezüglich der aufzudeckenden Fehler als sehr effektiv. Ein Produkt auf dem Markt einzuführen ohne dabei dieses Verfahren der Verifikation zu nutzen, lässt zu viel Raum für mögliche verbleibende Fehler. Die Folgen können von teuer über verheerend bis hin zu katastrophal sein.

Ein Beispiel dafür ist die Programmierung der Computer von Satelliten und Raumfahrzeugen

der NASA. Bei einem Schaden drohen Verluste in Höhe von Millionen USD. Bestes Beispiel war Ariane 501, die auf Grund eines *float overflows* bereits 37 Sekunden nach dem Start explodierte. Die Folge war ein direkter Schaden von 500 Millionen USD und ein indirekter von 7 Millionen USD, für 10 Jahre Forschung [7].

Die Summe der Codezeilen stieg mit der Zeit und den Projekten der NASA exponentiell an [1]. Während im Jahr 1977 für die Raumsonde Voyager lediglich 3 000 Zeilen Code benötigt wurde, waren es für die Raumsonde Cassini 1997 schon 32 000 und für die Internationale Raumstation (ISS) im Jahr 2000 sogar 1 700 000. Dieser Code wird von entsprechend mehr Entwicklern betreut, dennoch steigt die potentielle Gefahr von Fehlern an. Ein Fehler auf den Bordcomputern der ISS könnte wiederrum die gesamte Raumstation gefährden [1].

Eine Studie im Jahr 2002 über die wirschaftlichen Auswirkungen von Software-Fehler in den USA ergab jährliche Kosten von 22.2 Milliarden USD bis 59.5 Milliarde USD [8]. Diese Studie spiegelt dabei nur den Sachschaden wieder. Software-Fehler deren Auswirkungen katastrophale Dimension oder den Verlust von Menschenleben haben sind darin nicht enthalten.

Weitergehend ist statische Analyse eine klare Ergänzung zur dynamischen Analyse. Dynamische Tests benötigen ein für die Software lauffähiges System. Bei eingebetteten Systemen steht im Vergleich meist eine sehr geringe Zahl an Geräten zur Verfügung. Manchmal handelt es sich nur um ein einzelnes Gerät, wenn die Anschaffungkosten Millionen verursachen. Die Testläufe sind demnach sehr gering bzw. die Durchführung adäquat vieler Tests sehr zeitintensiv. Die Werkzeuge für die statische Analyse sind wiederrum Anwendungssoftware, welche auf herkömmlichen Systemen ausgeführt werden kann.

6 Zusammenfassung und Ausblick

In dieser Arbeiten wurden die grundlegenden Konzepte der statischen Analyse bezogen auf eingebette Systeme erläutert. Die Motivation war gerade für eingebette Systeme ein Verfahren vorzustellen, dass sehr genau verifiziert, zeitsparend und vollautomatisch ist. Die Theorie, die die statische Analyse ermöglicht, wurde anhand der drei Kozepte Kontroll- und Datenflussanalyse, sowie abstrakter Interpretation erklärt. Während abstrakte Interpretation das Prinzip vermittelt, wie statisch aus dem Quelltext Informationen gewonnen werden können, werden in der Kontroll- und Datenflussanalyse Verfahren verwendet, die diese dann auch algorithmisch verarbeiten.

Der Einblick in genutzte Werkzeuge auf dem Markt sollte deutlich gemacht haben, dass die Verfahren der statischen Analyse mittlerweile sehr weit in Form von Programmen umgesetzt wurden.

Die Analyse ist Teil jeder Programmentwicklung. Sei es die IDE, die bereits während der Programmierung darauf hinweist, dass deklarierte Variablen nie verwendet werden oder Compiler die unseren Code in weitaus efffizienteren Maschinencode optimieren.

Konsequente Softwarezuverlässigkeit wird für die Softwareentwicklung auch in Zukunft eine der großen Herausforderungen bleiben. Für die statische Analyse wird es eine Aufgabe sein mit der wachsenden Dimension der Softwareentwicklung zu skalieren. Objektorientierte, modulare oder nebenläufige Programmierung gehören dazu. Es verbleiben also die theoretischen Konzepte in die Praxis effektiver und effizienter umzusetzen und neue zu erschließen.

Literatur

[1] Webseite: Guillaume Brat. Experiences in the static analysis of embedded software. URL: http://ti.arc.nasa.gov/m/pub/1337/1337 (Brat).pdf, (Abgerufen: 01. Dezember 2009 00:34).

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[3] Webseite: Guillaume Brat, Willem Visser. Combining Static Analysis and Model Checking for Software Analysis. URL: http://ti.arc.nasa.gov/m/pub/archive/0307.pdf, (Abgerufen: 05. Dezember 2009, 16:39).

[4] Flemming Nielson Hanne Riis Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing, 1999.

[5] Matthew Sterling Hecht. *Global data-flow analysis of computer programs.* PhD thesis, Princeton University, Princeton, NJ, USA, 1973.

[6] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM.

[7] Ken Robinson. Flight 501 failure - a case study of errors. Technical report, University of New South Wales, 2009.

[8] RTI. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, Health, Social, and Economics Research, Mai 2002.

[9] O. Shivers. Control flow analysis in scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM.

[10] Webseite. The ASTRÉE Static Analyzer. URL: http://www.astree.ens.fr, (Abgerufen: 14. Januar 2010 21:35).

[11] Webseite. The MathWorks Deutschland - PolySpace Client for C/C++ 7.1. URL: http://www.mathworks.de/products/polyspaceclientc, (Abgerufen: 14. Januar 2010 21:39).

[12] Webseite. aiT Worst-Case Execution Time Analyzers. URL: http://www.absint.de/ait, (Abgerufen: 14. Januar 2010 21:45).