Verifiziertes Fiasco

Ein Projekt zur formalen Analyse und zum Beweisen der totalen Korrektheit des Mikrokern-Betriebssystems Fiasco

Hendrik Tews
wwwtcs.inf.tu-dresden.de/∼tews

Christoph Haase
www.inf.tu-dresden.de/∼s0158714

Technische Universität Dresden; Fakultät Informatik; D-01062 Dresden

12. Dezember 2004

Zusammenfassung

Dieser Artikel gibt einen kurzen Überblick über Softwareverifikation im allgemeinen und das an der TU Dresden verfolgte VFiasco-Projekt und die darin angewandten Techniken zur Softwareverifikation im Speziellen (VFiasco steht für *Verified Fiasco*).

1 Die neuesten Modewörter (Einleitung)

Fiasco [HH01] heißt das im Rahmen des DROPS-Projektes [HBB+98] an der Fakultät Informatik, TU Dresden entwickelte Mikrokernbetriebssystem (DROPS steht für *Dresden Real-Time Operating System*). Mikrokern bedeutet hierbei, dass Fiasco nur die *absolut notwendige* Kern-Funktionalität enthält. Dazu gehören zum Beispiel Prozesse und Threads, Speicherschutz (virtuelle Adressräume für Prozesse) und Kommunikation zwischen Prozessen, *jedoch nicht* Gerätetreiber für Festplatten und Graphikkarte. Fiasco ist nahezu vollständig in C++ geschrieben und umfasst weniger als 15.0000 Zeilen Quellcode (im Gegensatz zu 2,4 Millionen Zeilen in Linux 2.4 [Whe02]). Mit minimalen Laufzeiteinbußen kann man ein leicht modifiziertes Linux-System als Nutzerprogramm auf Fiasco laufen lassen [HHL+97]. Der Vorteil besteht hierbei darin, dass noch weitere Programme direkt auf Fiasco laufen können, zum Beispiel zum Signieren von Emails oder zum Abspielen von Videos. Ein überlastetes oder durch Vieren kompromitiertes Linux kann die anderen Anwendungen dabei nicht negativ beeinflussen. Das heißt, das Video läuft ruckelfrei, selbst wenn gerade parallel 100 Linux-Kerne compiliert werden. Und es ist wirklich der eigene Text signiert, und nicht der, den die im kompromittierten Linux gerade eingeloggten Borgs einem unterschieben wollen.

Das im Jahr 2000 an der TU Dresden gestartete VFiasco-Projekt (VFiasco steht für *verified Fiasco*) stellt sich zum Ziel, einige wesentliche Sicherheitseigenschaften des Fiasco-Quelltextes formal zu verifizieren. Im Projekt sollen insbesondere auch Methoden zur formalen Verifikation von C++-Programmen entwickelt werden. Formale Verifikation bedeutet hierbei, dass das Verhalten der Programme mit mathematischen Methoden untersucht wird.

Die dabei gewonnenen Ergebnisse gelten mit mathematischer Universalität. Zur Verifikation werden die Programme in eine geeignete mathematische Domäne abgebildet. Dort kann man mit mathematischen Beweisen[1] Eigenschaften der Programme zeigen.

Die Schwierigkeit bei der Verifikation von C++-Programmen liegt in Sprachbestandteilen, für die C und C++ berühmt-berüchtigt sind: Typumwandlungen (*type casts*), Sprungbefehle oder Tricks mit Stack- und Instruction-Pointer wie `setjmp/longjmp` [lon].

In diesem Artikel beschreiben wir einige Punkte der im VFisaco-Projekt entwickelten Semantik für Anweisungen von C++. Eine detaillierte Darstellung findet sich in [Tew00]. Unsere Semantik für Anweisungen basiert auf vergleichsweise sehr einfachen mathematischen Grundlagen. Sie erlaubt es jedoch, Sprungbefehle und auch `setjmp` und `longjmp` korrekt zu behandeln. Insbesondere können auch Sprünge *in einen Block hinein*, zum Beispiel in eine Schleife, behandelt werden. Das gestattet die Verifikation von *Duff's Device* [Duf04], einem Programm, das selbst erfahrenen C- oder C++- Programmierern das Gruseln lehrt. Unsere Formalisierung der C++-Datentypen, die auch die Behandlung von Typumwandlungen gestattet, passt nicht in diesen Artikel, siehe [HT03].

Bevor wir jedoch zur Semantik von Sprungbefehlen gelangen, geben wir im nächsten Abschnitt einen kleinen Überblick über verschiedene Methoden zur Qualitätssicherung von Software. In Abschnitt 3 stellen wir unsere Semantik für Anweisungen von C++ vor. Abschnitt 4 zeigen wir, wie man die Semantik benutzen kann, um die Korrektheit von Duff's Device zu beweisen.

2 Die vier Stufen zum Gral der fehlerfreier Software

In diesem Abschnitt geben wir einen Überblick über verschiedene Methoden und Ansätze, die Qualität von Software zu erhöhen. Das Ziel aller dieser Methoden ist dabei, sich selbst, den Kunden oder das Bundesamt für Sicherheit in der Informationstechnik mehr oder weniger davon zu überzeugen, dass ein gegebenes Programm bestimmte Eigenschaften und Anforderungen erfüllt. Diese Eigenschaften und Anforderungen werden allgemein als *die Spezifikation* bezeichnet. Eine Spezifikation kann in jeder Form vorliegen, zum Beispiel auch in natürlicher Sprache. Sie kann aber auch in logischen Formeln abgefasst sein. In diesem Fall sprechen wir von einer *formalen* Spezifikation. Durch die Verwendung einer künstlichen Sprache gibt es keine Auslegungsdifferenzen für eine formale Spezifikation. Man kann sich höchstens noch darüber streiten, ob die Spezifikation auch wirklich die gewünschten Eigenschaften enthält.

Es gibt verschiedene Methoden, um zu überprüfen, ob ein Programm eine Spezifikation erfüllt. Die Methoden unterscheiden sich hinsichtlich ihrer Kosten und der Sicherheit, die sie dafür bieten, dass die Spezifikation danach vom Programm auch eingehalten wird. Allgemein kann man sagen, dass erfolgversprechende Methoden um ein vielfaches teurer sind, als die Erstellung des Programmes selbst. Sie werden heutzutage nur angewendet, wenn es um sehr viel Geld oder um die Sicherheit von Menschen geht. Die im VFiasco-Projekt verwendete Methode, nämlich *mechanische Verifikation mit Hilfe von denotationeller Semantik*, ist möglicherweise die aufwendigste Methode, die aber auch die größten Garantien liefert. Ein

[1] Ein korrekter mathematischer Beweis garantiert die universelle Gültigkeit der bewiesenen Aussage im Gegensatz zu juristischen, philosophischen oder soziologischen Beweisen, wo der Wahrheitsgehalt der Aussage je nach Instanz, moralischem Standpunkt oder den zugrunde gelegten Statistiken schwankt.

Hauptziel von VFiasco ist, erstmalig nachzuweisen, dass man ein Betriebssystem überhaupt auf diese Art und Weise verifizieren kann.

2.1 Testen

Jeder hat es schon getan: Einen Probelauf eines Programmes um zu sehen, ob die erwarteten Ausgaben wirklich produziert werden. Leider kann man mit Testen nur Fehler finden, nie aber Fehlerfreiheit nachweisen. Die Testmethoden reichen vom einfachen Probieren bis zu systematisch ausgearbeiteten Tests, die während der Programmentwicklung und -wartung regelmäßig durchgeführt werden [Mye01, JC00]. Mit statistischen Auswertungen lässt sich dann auch die Anzahl der noch nicht gefundenen Fehler schätzen.

2.2 Statische Programmtests

Unter statischen Tests versteht man besondere Checks, die man am Quellcode des gesamten Programmes vornehmen kann. Zum Beispiel generiert `gcc` mit der Option `-Wall` Warnungen für Variablenbenutzungen vor deren Initialisierung. Welche statischen Tests sinnvoll sind, hängt in starkem Maße vom betrachteten Programm ab. In Betriebssystemen muss zum Beispiel jeder Pointer, der vom Nutzer kommt, besonders geprüft werden, bevor er für irgendeine andere Operation verwendet werden kann. Mit Hilfe solch einfacher Regeln und entsprechenden Tools zum automatischen Checken fand zum Beispiel Engler mit seinen Kollegen 132 Bugs in den Quellen von Linux 2.3.99 [ECCH00].

2.3 Model Checking

Model Checking zählt zu den automatischen Methoden. Das heißt, das der Ingenieur zwar das Programm und dessen Spezifikation selbst anfertigt, dann aber nur einen Knopf drückt und solange Kaffee trinkt, bis das Verifikationsprogramm antwortet „gilt“ oder „gilt nicht.“ Automatische Methoden zur Verifikation von Software haben mit zwei Problemen zu kämpfen: Erstens müssen potentiell alle möglichen Zustände des Programms betrachtet werden, das heißt jeder Punkt im Programmablauf mit allen möglichen Variablenbelegungen. Das führt zu einer expotentiell verlaufenden Zustandsexplosion (jedes zusätzliche benutzte Bit in einer Variablen führt zur Verdopplung des Zustandsraumes), die die real begrenzten Ressourcen (Zeit und Speicher) schnell ausschöpft. Das zweite Problem ist, dass eine Reihe interessanter Fragen, zum Beispiel ob das Programm für alle Eingaben terminiert, prinzipiell von Rechnern nicht automatisch beantwortet werden können.[2]

Model Checking ist der Versuch, das Beste aus dieser Situation zu machen. Dafür wird aus dem zu verifizierenden Programm zunächst durch Abstraktion ein Modell gewonnen. Wichtig ist, dass dieses Modell einen endlichen Zustandsraum hat. Alle Variablen haben also begrenzte Wertebereiche, wie im richtigen Programm. Bei der Abstraktion werden alle für die Verifikation unwesentlichen Details vernachlässigt. Für die Implementation eines

[2] Solche Probleme werden *unberechenbar*, oder, falls es sich um ja-nein Fragestellungen handelt, *unentscheidbar* genannt. Das berühmteste Beispiel ist das sogenannte Halteproblem, das darin besteht, für ein beliebiges Programm mit einer beliebigen Eingabe vorherzusagen, ob das Progamm terminiert. Es könnte zwar einen Hellseher geben, der das kann. Es läßt sich jedoch beweisen, dass es kein Computerprogramm mit dieser Fähigkeit geben kann [Hal].

Kommunikationsprotokolls würde das Modell typischerweise nur Auf- und Abbau des Kommunikationskanals enthalten. Die zu übertragenden Nachrichten würden genau wie Tests auf unzulässige Parameter und Fehler- oder Loggingausgaben *wegabstrahiert* werden.

Diejenigen Eigenschaften des Originalprogramms, die sich als Prädikat über den Zuständen des Modells formulieren lassen, können nun vollautomatisch durch Durchmusterung aller Zustände überprüft werden. Das übernimmt ein *Modelchecker* wie zum Beispiel Spin [Hol03] oder SMV [McM93]. Durch eine geniale Repräsentation der Zustandsmenge lassen sich Modelle mit einem gigantischen Zustandsraum von bis zu 10^{100} Elementen[3] bearbeiten.

Falls der Modelchecker die Eigenschaft nicht beweisen kann, gibt er immer einen fehlerhaften Zustand aus und die Operationen, mit denen man diesen Zustand erreichen kann. Damit kann man am realen Programm überprüfen, ob es sich wirklich um einen Fehler handelt oder ob das Problem erst durch die Abstraktion vom Programm zum Modell entstanden ist. Im zweiten Fall muss man Abstraktion und Modell modifizieren.

Umgekehrt kann es natürlich auch passieren, dass erst durch die Abstraktion zum Modell die Eigenschaft beweisbar wurde und dass sie im originalen Programm gar nicht gilt. Es kann auch sein, dass die Eigenschaft gilt, das Programm aber immer wegen eines Fehlers im wegabstrahierten Teil abstürzt. Das zeigt, ein prinzipielles Problem bei der Anwendung formaler Methoden: 100-prozentige Sicherheit kann man nie erreichen. Zwar gelten die mit mathematischen Methoden erreichten Resultate. Man kann aber nicht ausschließen, dass sich im Beweis ein Fehler eingeschlichen hat. Ein Beweis zur Fehlerfreiheit des Beweises[4] ist zwar möglich, nur leidet der Überbeweis wieder am gleichen Problem. Deshalb führen auch mathematischen Methoden letztendlich nur zu einem (allerdings enorm) vergrößerten Vertrauen in die Gültigkeit der Spezifikation.

2.4 Verifikation mit operationaler und denotationeller Semantik

Bei der Verifikation mit Hilfe einer Semantik bestimmt man zuerst die *semantische Domäne*. Das ist eine geeignete Menge mathematischer Objekte, die das Verhalten aller möglichen Programme adäquat abbilden können. Die Elemente der semantischen Domäne sind meist sehr komplex. Benutzt werden zum Beispiel Funktionale (das heißt Funktionen, die Funktionen auf Funktionen abbilden), Transitionssysteme mit Funktionen als Zustände oder Scott-Domänen.[5] Zusammen mit der semantischen Domäne wählt man eine Abbildung, die *Semantikfunktion*, die jedes Programm und jedes Programmfragment in ein Element der Domäne abbildet. Wenn die Semantik wirklich benutzt werden soll, muss die Semantikfunktion

[3] 10^{100} ist unglaublich viel. Zum Vergleich: Die Zahl der Sterne im sichtbaren Bereich des Universums ist etwa 10^{23}, die der Elementarteilchen etwa 10^{80}. Eine Kugel von 10^{100} Siliziumatomen hat (bei Erddichte) einen Radius von 25 Gigaparsec (10^{27}m), 10 mal mehr als der Radius des sichtbaren Teils des Universums. Andererseits erreicht man 10^{100} Zustände schon mit Variablen von insgesamt reichlich 300 Byte.

[4] Dieser Ansatz wird tatsächlich verfolgt, zum Beispiel in Coq [Coq]. Den Beweis selber kann man mit beliebiger, auch fehlerbehafteter Software erstellen. Anschließend wird der Beweis aber von einem nur wenige hundert Zeilen umfassenden Modul auf Richtigkeit Überprüft.

[5] Scott-Domänen sind so komplex, dass Peter Mosses dem Fachpublikum in [Mos90] allen Ernstes vorschlägt, man solle sich nicht mit den Details beschäftigen. Für die unerschrockene Leserin: Scott Domänen sind induktiv-vollständige Ordnungsrelationen (jede gerichtete Menge hat ein Supremum), die algebraisch sind (jedes Element ist das Supremum einer gerichteten Menge kompakter Elemente) und die nur abzählbar viele kompakte Elemente besizten. Ein Element x ist kompakt, wenn jede gerichtete Menge, deren Supremum größer als x ist, bereits ein Element enthält, das größer als x ist.

automatisiert sein, das heißt, es muss eine Art Compiler geben, der Programmfragmente in ihre Semantik übersetzt.

Steht die semantische Domäne fest, übersetzt man das zu verifizierende Programm in ein Element der semantischen Domäne. Spezifikationen und Korrektheitsbeweise kann man dann mit allen in der Mathematik zur Verfügung stehenden Mittel darstellen und entwickeln.

Operationale Semantik Bei operationaler Semantik besteht die semantische Domäne aus Transitionssystemen. Ein Transitionssystem ist ein Graph bestehend aus Knoten und Pfeilen (den Transitionen) zwischen den Knoten. Bei operationaler Semantik bestehen die Knoten aus einem Programmzustand und einem Stückchen Quellprogramm. Der Programmzustand enthält alles für die Programmabarbeitung wesentliche, zum Beispiel die Werte aller Variablen. Eine Transition $(s,p) \longrightarrow\!\!\!\!\!\rightarrow (r,q)$ bedeutet, dass eine teilweise Abarbeitung des Programmes p im Startzustand s zu einem Speicherzustand r führen kann, in dem noch das Restprogramm q abgearbeitet werden muss.

Die Transitionen werden meist durch ein Regelsystem beschrieben. Eine Regel

$$\frac{\text{Voraussetzung}_1 \quad \text{Voraussetzung}_2 \quad \text{Voraussetzung}_3}{\text{Ziel}} \text{ Seitenbedingung}$$

bedeutet dabei, dass das Ziel ableitbar ist, vorausgesetzt, die Seitenbedingung ist erfüllt und die Voraussetzungen lassen sich mit dem gleichen Regelsystem (in endlicher Weise) ableiten. Typische Regeln für eine idealisierte imperative Programmiersprache sehen wie folgt aus.

$$\frac{}{\langle s, x := a\rangle \longrightarrow \langle s|_{x\mapsto a}, \varepsilon\rangle}$$

$$\frac{}{\langle s,\ \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\rangle \longrightarrow \langle s, p_1\rangle} \quad [\![b]\!](s) = \mathit{wahr}$$

$$\frac{\langle s, p_1\rangle \longrightarrow \langle s', p_1'\rangle}{\langle s, p_1; p_2\rangle \longrightarrow \langle s', p_1'; p_2\rangle} \qquad \frac{\langle s, p_1\rangle \longrightarrow \langle s', \varepsilon\rangle}{\langle s, p_1; p_2\rangle \longrightarrow \langle s', p_2\rangle}$$

Die erste Regel (ohne Voraussetzungen) beschreibt die Zuweisung, deren Effekt nur darin besteht, den Wert der Variablen im aktuellen Zustand zu ändern.[6] Dabei steht ε für das leere Programm. Die zweite Regel für die `if`-Anweisung besagt, dass man das gesamte `if` zu p_1 vereinfachen kann, vorausgesetzt, die Bedingung b ergibt, im Zustand s ausgewertet, wahr. (Natürlich gibt es noch eine Regel für $[\![b]\!](s) = \mathit{falsch}$, für die wir den Platz aber hier sparen.) Die letzten beiden Regeln behandeln die sequentielle Komposition von Anweisungen. Dabei wird von links abgearbeitet, solange, bis die Anweisung p_1 aufgebraucht ist.

Das Regelsystem einer operationalen Semantik gestattet, Beweise zur Termination oder zum Inhalt des Endzustandes (in Abhängigkeit vom Anfangszustand) eines Programmlaufes zu führen. Für eine realistische Programmiersprache mit Seiteneffekten, Ausnahmen (*exceptions*) und Anweisungen wie `break` und `return` ist das operationale Regelwerk natürlich etwas komplizierter.

[6] $s|_{x\mapsto a}$ steht für einen Zustand, der der Variablen x den Wert a zuordnet, ansonsten aber identisch mit s ist.

Denotationelle Semantik Die semantische Domäne für denotationelle Semantik ist meist eine Menge von Funktionen. Jedes Programmfragment bekommt eine solche Funktion als Semantik zugeordnet. Als Beispiel betrachten wir die Menge aller Programmzustände S (wie aus dem vorherigen Abschnitt) und dazu die Menge aller Funktionen $S \rightharpoonup S_\perp$. Dabei enthält $S_\perp$ genau ein zusätzliches Element, nämlich $\perp$, das nicht in S enthalten ist. Das zusätzliche Element $\perp$ benötigt man, damit die Semantik auch Programmen, die für einen speziellen Anfangszustand nicht terminieren oder andersweitig abstürzen, einen Resultat (nämlich $\perp$) zuordnen kann.

Hier sind ein paar typische Beispiele für semantische Gleichungen (die doppelten eckigen Klammern $[\![-]\!]$ stehen dabei für die semantische Funktion):

$$
\begin{aligned}
[\![x := a]\!](s) &= s|_{x \mapsto a} \\
[\![\texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2]\!](s) &= \begin{cases} [\![p_1]\!](s) & \text{falls } [\![b]\!](s) = \mathit{true} \\ [\![p_2]\!](s) & \text{sonst} \end{cases} \\
[\![p_1; p_2]\!](s) &= \big([\![p_2]\!] \circ [\![p_1]\!]\big)(s) \quad = \quad [\![p_2]\!]\big([\![p_1]\!](s)\big)
\end{aligned}
$$

Die Semantik der Zuweisung ist also eine Funktion, die jeden Zustand s auf den bei x geänderten Zustand $s|_{x \mapsto a}$ abbildet.

Denotationelle Semantik wird häufig in einer *kompositionalen* Art und Weise definiert. Das bedeutet, dass die Semantikdefinition einer komplexer Anweisung mit Hilfe der Semantik ihrer einfacheren Bestandteile definiert wird. Die Bestandteile der `if`-Anweisung sind zum Beispiel die Bedingung b und die Programmfragmente p_1 und p_2. In der zweiten Gleichung wird deren Semantik als bekannt vorausgesetzt. Damit bildet die Semantik von `if` einen Zustand s entweder auf $[\![p_1]\!](s)$ oder $[\![p_2]\!](s)$ ab, je nachdem, welchen Wert b im Zustand s annimmt.

Die Semantik sequentieller Komposition ist einfach die Komposition der Semantiken. Die Semantik von potentiell nicht terminierenden Schleifen und von rekursiven Funktionen wird traditionell mit recht komplizierten Methoden definiert.[7]

Betrachten Sie das folgende Programm.

```
y := 0;
if y = 0 then x = 1;
else x = 2;
```

Mit den oben aufgeführten Gleichungen kann man dessen Semantik bestimmen als diejenige Funktion, die jeden Zustand s auf $s|_{x \mapsto 1}$ abbildet.

Für richtige Programmiersprachen ist die semantische Funktion natürlich um einiges komplizierter. Auch wenn Programm und Spezifikation klein sind, ist die Verifikation schnell zu komplex, um sie mit Papier und Stift zu bewältigen. Die eigentliche Arbeit besteht meist

[7] Wer es genau wissen will: Die Funktionen $S \rightharpoonup S_\perp$ bilden eine induktiv-vollständige Ordnung (siehe Fußnote 5), und zwar gilt $f \leq g$ wenn g für mehr Zustände definiert ist und wenn außerdem $f(s) = g(s)$ für die Zustände s gilt, für die f und g beide definiert sind. Eine monotone Funktion auf einer induktiv-vollständigen Ordnung hat immer einen kleinsten Fixpunkt, der sich endlich approximieren lässt, falls die Funktion auch stetig (im ordnungstheoretischen Sinn) ist. Die Körper von `While`-Schleifen und von rekursiven Funktionen werden in entsprechende monotone Funktionen übersetzt. Als Semantik nimmt man dann den kleinsten Fixpunkt.

im geduldigen Überprüfen vieler kleiner Seitenbedingungen, wie dem Einhalten von Arraygrenzen. Diese Arbeit überträgt man am besten entsprechender Software. Nur an wenigen Stellen, wie zum Beispiel bei Terminationsbeweisen, ist wirklich Kreativität gefragt. *Softwareverifikation ist komplex, aber nicht kompliziert.* Die Arbeit beim Verifizieren kann man nur schlecht mit der eines Mathematikers vergleichen: Der Mathematiker sucht nach Beweisen, die (noch) niemand kennt. Bei der Softwareverifikation gibt es immer einen der (hoffentlich) erklären kann, warum es funktioniert: Den Programmierer.

Natürlich besteht auch bei der Arbeit mit operationaler oder denotationeller Semantik die Gefahr, dass die Verifikationsergebnisse nicht auf die Wirklichkeit zutreffen. Zum Beispiel könnte die semantische Domäne inkonsistent sein, das heißt die Ableitung beliebiger Ergebnisse (und nicht nur der richtigen) ermöglichen. Bei der Übersetzung in die Semantik könnte sich ein Fehler eingeschlichen haben. Die bei der Verifikation eingesetzte Software könnte einen Fehler haben (oder der Gutachter, der den Beweis nachrechnet einen Fehler übersehen).

Als die größte Gefahr betrachten die meisten Autoren heute jedoch die Verifkation von Quelltexten. Denn die Ergebnisse, die von den Quelltexten abgeleitet werden, gelten nur, wenn auch der Compiler, der das ausführbare Programm generiert hat, keinen Fehler gemacht hat. Wie schon weiter oben erwähnt, gilt auch hier, dass eine Verifikation mit operationaler oder denotationeller Semantik keine absolute Sicherheit gewährleisten kann, sondern nur das Vertrauen in die Fehlerfreiheit erhöht. Um die Relationen zu wahren, müssen wir jedoch das Folgende klarstellen: Eine sorgfältig durchgeführte Verifikation des Quelltextes bietet tausend Mal mehr Sicherheit als alle anderen Methoden zur Qualitätssicherung von Software. Sie übertrifft auch bei weitem das Maß an Sicherheit, das TüV-geprüfte technische Systeme haben.

3 Richtig springen (Eine denotationelle Semantik für Goto)

In diesem Abschnitt beschreiben wir einige Aspekte einer denotationellen Semantik für C oder C++, die es uns gestattet, Programme wie Duff's Device zu verifizieren. Diese Semantik wurde im Rahmen des VFiasco-Projektes entwickelt und zuerst in [Tew00] der Öffentlichkeit vorgestellt. Sie ist eine Weiterentwicklung der im LOOP-Projekt [LOO] entwickelten Semantik für Java [HJ00].

3.1 Zusammentun ohne zu mixen (Disjunkte Vereinigung)

Im Folgenden benötigen wir den Begriff der *disjunkten Vereinigung,* der leider nicht zum Allgemeingut gehört. Man erhält die disjunkte Vereinigung zweier Mengen M und N, in Zeichen $M \uplus N$, indem man vor der eigentlichen Vereinigung dafür sorgt, dass alle Elemente in M und N verschieden sind. Damit ist die disjunkte Vereinigung immer genau so gross wie die Summe der Größen von M und N. Außerdem ist ein Element x, das wir in $M \uplus N$ finden *entweder* aus M *oder* aus N, niemals aus beiden.

Disjunkte Vereinigungen werden beim Programmieren relativ häufig benutzt, zum Beispiel treten Sie in Pascal und Modula-2 unter dem Pseudonym *Variantenrecords* auf [Mod]. Benutzt man den Union-Typ von C mit einem Tag hat man auch eine disjunkte Vereinigung. Die populärste Anwendung ist sicherlich der aus wenigstens 31 disjunkt vereinigten

Komponenten bestehende Typ `XEvent` aus der X-Programmbibliothek (oder aber dessen Windows-Äquivalent).

Man kann die disjunkte Vereinigung definieren, indem man die Elemente von M und N entsprechend markiert, zum Beispiel

$$M \uplus N \quad \stackrel{\text{def}}{=} \quad \{(m, 0) \mid m \in M\} \ \cup \ \{(n, 1) \mid n \in N\}$$

Im Folgenden verwenden wir allerdings bedeutungstragende Namen als Marken. Zum Beispiel steht $U \stackrel{\text{def}}{=} \overset{\mathit{links}:}{X} \uplus \overset{\mathit{mitte}:}{Y} \uplus \overset{\mathit{rechts}:}{Z}$ für die disjunkte Vereinigung der drei Mengen X, Y und Z wobei die Elemente aus X mit *links* markiert wurden, die aus Y mit *mitte* und so weiter. Die Marken verwenden wir gleichzeitig als Funktionen. So ist $\mathit{links}(x) \in U$ für $x \in X$.

3.2 Komplexe Zustände

Ohne weiter ins Detail zu gehen, betrachten wir wieder eine Menge von Zuständen S, wobei jeder Zustand auf irgendeine Art und Weise alle Werte aller benutzten Variablen enthält. Jetzt bilden wir die folgende disjunkte Vereinigung aus 7 Komponenten:[8]

$$\mathit{Res} \quad = \quad \overset{\mathit{ok}:}{S} \uplus \overset{\mathit{break}:}{S} \uplus \overset{\mathit{case}:}{(S \times \mathbb{Z})} \uplus \overset{\mathit{default}:}{S} \uplus \overset{\mathit{goto}:}{(S \times \mathbb{L})} \uplus \overset{\mathit{fail}:}{\mathbf{1}} \uplus \overset{\mathit{hang}:}{\mathbf{1}}$$

Bevor wir das erläutern sagen wir noch kurz, wohin die Reise geht: die Funktionen $\mathit{Res} \longrightarrow \mathit{Res}$ bilden die semantische Domaine unserer Semantik! Das heißt jedes Programmfragment bekommt als Semantik eine solche Funktion zugeordnet.

Elemente von *Res* nennen wir im Folgenden *komplexe* Zustände (im Gegensatz zu den Elementen von S). Jede der sieben Komponenten von *Res* steht für ein mögliches Resultat einer Anweisung in C oder C++:[9] Die *ok*-Komponente steht für normale Termination einer Anweisung, nach der die Programmausführung mit der darauf folgenden Anweisung fortgesetzt wird. Die anderen Komponenten bezeichnen eine sogenannte abnormale Termination, die dazu führt, dass eine Reihe der (im Programm textuell) folgenden Anweisungen übersprungen wird. Die Marken *break*, *case*, *default* und *goto* stellen dabei zeitweilige Abnormalitäten dar, die an späterer Stelle, zum Beispiel am Ende einer Schleife oder einem Label, wieder in den normalen *ok*-Modus transferiert werden können. Deshalb kapseln diese Abnormalitäten alle einen Zustand, nämlich den Zustand, mit dem die Ausführung fortgesetzt werden soll.

Die Marken *fail* und *hang* stehen für einen Programmabsturz, der durch einen Typfehler (*fail*) oder durch eine nicht terminierenden Schleife (*hang*) verursacht werden kann. Die Abnormalitäten werden wie folgt benutzt.

break ist das Resultat einer `break`-Anweisung.[10] Ein mit *break* markierter Zustand ist der Zustand *vor* dem Ausführen der `break`-Anweisung, mit dem die Ausführung hinter der Schleife fortgesetzt wird.

[8] Das ist die vereinfachte Definition aus [Tew00], in der einige die Komponenten, wie zum Beispiel für `continue` und `longjmp`, fehlen

[9] Dabei haben wir geschummelt und der Einfachheit halber für diesen Artikel die drei Komponenten für `return`, `continue` und `longjmp` weggelassen.

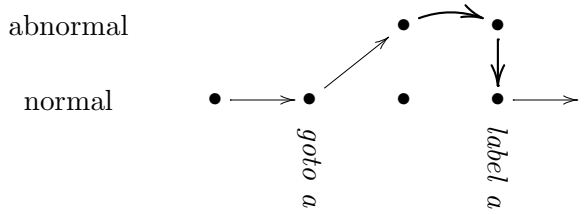
[10] Die `break`-Anweisung verläßt auf direktem Wege die umschließende `while` oder `for`-Schleife und setzt mit der ersten Anweisung nach der Schleife fort.

case Mit *case* wird immer ein Paar aus einem Zustand und einer Zahl markiert. Die *case*-Abnormalität wird benutzt, um die Anweisungen zwischen dem `switch` und dem richtigen `case` zu überbrücken.

default Mit der *default*-Abnormalität gelangt man vom `switch` zum `default`-Label (so es eines gibt).

goto ist das Resultat einer `goto` Anweisung, sie überbrückt die Ausführung bis zu einem passenden Label.

Die folgende Graphik verdeutlicht die Verwendung der Abnormalitäten am Beispiel von `goto`:



Wir starten in einem mit *ok* markierten komplexen Zustand, die ersten Anweisungen werden ausgeführt. Das Resultat des `goto`-Befehls ist ein mit *goto* markierter Zustand. Die drauffolgenden Anweisungen werden inspiziert, der mit *goto* gekapselte Zustand wird jedoch nicht verändert. Das passende Label transformiert die *goto*-Abnormalität schließlich wieder in ein *ok* und die darauffolgenden Anweisungen werden wieder ausgeführt.

3.3 Effekte fangen (Semantik einfacher Anweisungen)

Im Folgenden definieren wir beispielhaft die Semantik einiger Anweisungen. Die nicht besprochenen Details sind in [Tew00] zu finden. Wie schon gesagt, wird die Semantik aller Anweisungen und Programmfragmente als eine Funktion $Res \longrightarrow Res$ dargestellt. Am einfachsten ist die Semantik von `break` und `goto`:

$$[\![\texttt{break}]\!] = \left| \begin{array}{lcl} ok(s) & \mapsto & break(s) \\ x & \mapsto & x \end{array} \right.$$

$$[\![\texttt{goto } l]\!] = \left| \begin{array}{lcl} ok(s) & \mapsto & goto(s, l) \\ x & \mapsto & x \end{array} \right.$$

In diesen Gleichungen benutzen wir Mustervergleich (*pattern matching*), wie in funktionalen Programmiersprachen üblich: Fall das Argument von $[\![\texttt{break}]\!]$, also der komplexe Zustand in dem die `break`-Anweisung startet, mit *ok* markiert ist, wird der enthaltene Zustand s mit *break* markiert. Anderenfalls passiert nichts. Für die Semantik von `goto` speichert die *goto*-Abnormalität auch das Ziellabel.

Betrachten wir nun die Semantik eines Labels:

$$[\![l :]\!] = \left| \begin{array}{lcl} goto(s, l) & \mapsto & ok(s) \\ x & \mapsto & x \end{array} \right.$$

Beachten Sie, dass von $[\![l :]\!]$ nur *goto*-Abnormalitäten mit dem richtigen Label transformiert werden. Andere *goto*-Abnormalitäten bleiben unverändert.

Die Semantik von Zuweisungen muss so definiert werden, dass sie nur einen Effekt für *ok*-markierte Zustände hat.

$$
\begin{array}{rcl}
[\![v = expr]\!] & = & \left| \begin{array}{lcll} ok(s) & \mapsto & \left\{ \begin{array}{ll} ok(s|_{v \mapsto i}) & \text{falls } [\![expr]\!](s) = ok(i) \\ fail & \text{sonst} \end{array} \right. \\ x & \mapsto & x \end{array} \right. \\
[\![st_1; st_2]\!] & = & \left| \; x \mapsto [\![st_2]\!]\big([\![st_1]\!](x)\big) \right.
\end{array}
$$

Bei der Variablenzuweisung ist zu beachten, dass die Berechnung des Ausdruckes *expr* wegen eines Typfehlers fehlschlagen kann.[11] Sequentielle Komposition wird wieder auf Funktionskomposition in der semantischen Domäne abgebildet. Dadurch wird sichergestellt, dass eine Abnormalität an allen Anweisungen vorbei kommt und schließlich diejenige findet, die sie wieder zu *ok* transformiert.

3.4 Abnormalitäten durchreichen (Semantik zusammengesetzter Anweisungen)

In diesem Abschnitt weisen wir auf ein paar Besonderheiten der Semantik von `if` und `switch`-Anweisungen sowie von Schleifen hin. Die semantischen Gleichungen in diesem Artikel zu diskutieren, würde zu weit führen. Der interessierte Leser findet sie in [Tew00].

Um für die Besonderheiten der Semantik von zusammengesetzten Programmen ein Gefühl zu bekommen, betrachten wir das folgende Programmfragment:

```
if(a == 0)
  goto lab;
else
  ;
switch(b){
  case 0: if(c == 0)
     lab:    d = 1;
          else
  case 1:    d = 2;
}
```

Das Fragment wird ein korrektes C oder C++ Programm wenn Variablendeklarationen und eine `main`-Funktion hinzugefügt werden. Der Effekt ist wie folgt: Falls `a` gleich Null ist, wird direkt in den `then`-Zweig zu der Zuweisung `d = 1;` gesprungen. Nach diesem Sprung geht die Abarbeitung normal weiter, das heißt das `else` führt dazu, dass das `if` und damit das `switch` verlassen werden. Falls `a` jedoch nicht Null ist, dann kann die `switch`-Anweisung dazu führen, dass in den `else`-Zweig gesprungen wird.

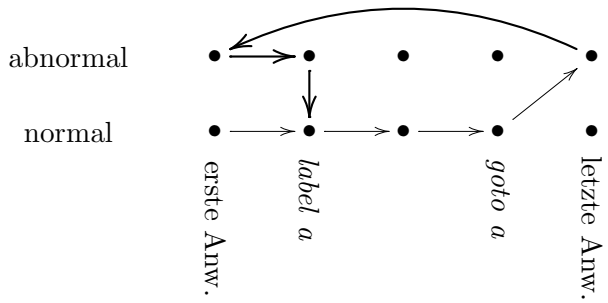
Für die Semantik der `if`-Anweisung hat das die folgenden Konsequenzen: Falls nach dem Ende des `then`-Teiles ein mit *ok* markierter Zustand vorliegt wird der `else`-Zweig ignoriert. Falls aber eine Abnormalität vorliegt, muss nach dem `then`-Zweig auch die Semantik des `else`-Zweiges evaluiert werden.

[11] Ausdrücke mit Seiteneffekten können leicht integriert werden. Damit und mit der Semantik der Ausdrücke wollen wir jedoch keinen Platz verschwenden.

Für zusammengesetzte Anweisungen gilt allgemein, dass enthaltene Ausdrücke (wie die Bedingung im `if` oder im `switch`) ignoriert werden, wenn die Anweisung mit einer Abnormalität betreten wird. Die Abnormalität muss dann aber an alle Anweisungsblöcke weitergegeben werden, und zwar in der Reihenfolge, in der sie im Quellprogramm stehen.

Traditioneller Weise wird die Semantik von `while` Schleifen so wie die von rekursiven Funktionen mit Hilfe einer Fixpunktkonstruktion auf der induktiv-vollständigen Ordnung der partiellen Funktionen bestimmt. Für `while`-Schleifen erhält man eine äquivalente, aber einfachere Semantik durch die Betrachtung von *Terminationspunkten*. Ein Terminationspunkt ist eine Anzahl von Iterationen der Bedingung und des Schleifenkörpers sodass nach dieser Anzahl von Iterationen die Schleife terminieren würde. Das heißt, es muss entweder eine Abnormalität vorliegen[12] oder die Bedingung muss falsch ergeben haben. Die Semantik der `while`-Schleife testet nun zuerst die Existenz von Terminationspunkten für den gegebenen Anfangszustand. Falls solche existieren ist die Semantik der gesamten Schleife identisch mit der Iteration bis zum kleinsten Terminationspunkt. Falls keine solchen Punkte existieren ergibt die Semantik *hang*. Eine so definierte Semantik gestattet zwar, für jede beliebige Schleife, zu bestimmen, ob sie terminiert oder nicht. Das Halteproblem ist dennoch nicht gelöst, da die Semantik von `while`-Schleifen nicht berechenbare Funktionen ergibt, das heißt, es kann keine Programme geben, die die Semantikfunktion berechnen.[13]

Zur Modellierung von Rückwärtssprüngen mit `goto` ist es wichtig, dass in C und C++ Sprungmarken einen Sichtbarkeitsbereich haben, nämlich genau den Funktionskörper in dem sie stehen. Dadurch wird verhindert, dass man von einer Funktion in eine andere springen kann. Für die Semantik von Rückwärtssprüngen legt man eine `goto`-Schleife um den gesamten Funktionskörper. Diese Schleife iteriert den Körper so lange, bis er nicht mehr mit einer *goto*-Abnormalität terminiert.



In diesem Bild springt die fünfte Anweisung zur zweiten zurück. Nach dem Sprung geht die Auswertung ersteinmal *vorwärts* weiter, allerdings wird durch die `goto`-Abnormalität der aktuelle Zustand nicht verändert. Schliesslich beginnt die Auswertung mit der `goto`-Abnormalität wieder von vorn und das Label wird erreicht.

In diesem Abschnitt haben wir zugleich sehr alte und brandneue Ideen vorgestellt. Der Zenit an Forschungsaktivität zu denotationeller Semantik ist sicherliche schon seit einigen Jahren überschritten. Eine denotationelle Semantik, die die Sprungbefehle von C und C++

[12] Genauer gesagt eine von `continue` verschiedene Abnormalität, denn eine `continue` Abnormalität wird am Ende des Schleifenkörpers zu `ok` transformiert.

[13] Den Test zur Existenz von Terminationspunkten kann man nicht so programmieren, dass er immer terminiert. Genau das wäre aber für eine Lösung des Halteproblems erforderlich.

```
void copy(char * to, char * from, int count){
  int rounds= count / 8;
  switch(count % 8){
  case 0: while(rounds-- > 0){  *to++ = *from++;
  case 7:                       *to++ = *from++;
  case 6:                       *to++ = *from++;
  case 5:                       *to++ = *from++;
  case 4:                       *to++ = *from++;
  case 3:                       *to++ = *from++;
  case 2:                       *to++ = *from++;
  case 1:                       *to++ = *from++;
} }                            };
```

Abbildung 1: Duff's device

korrekt behandelt, insbesondere die Sprünge *in* tiefer verschachtelte Blöcke *hinein*, war bisher jedoch unbekannt. Außerdem ist die von uns entwickelte Semantik ungewöhnlich einfach. Bis auf disjunkte Vereinigungen stammen alle Grundlagen aus der Schulmathematik. Domänentheorie oder Continuations, die sonst die Grundlagen denotationeller Semantik bilden, werden nicht benötigt.

4 Eigentlich unmöglich, aber trotzdem richtig (Ein Korrektheitsbeweis für Duffs Device)

Abbildung 1 zeigt Duff's Device [Duf04]. Es besteht aus einer achtfach aufgefalteten Schleife zum Kopieren, *ohne extra Kode zum Behandeln überzähliger Elemente.* Das Programm ist korrektes C und C++. Falls die Anzahl `count` zu kopierender Elemente nicht durch acht teilbar ist, wird mittels *switch* die Dekrementierung von *rounds* übersprungen und die überzähligen Elemente werden im ersten partiellen Schleifendurchlauf kopiert.

Abbildung 1 stimmt nicht ganz mit dem originalen Duff's Device überein. Tom Duff hat sein Device erfunden um Daten auf einen Port zu kopieren (das heißt die Zieladresse blieb konstant), wozu es auf der damaligen Architektur keine Assemblerprimitive gab. Außerdem hat er eine `do while`-Schleife verwendet.

Zur Verifikation von Duff's device benutzen wir den interaktiven Theorembeweiser PVS [ORR+96, PVS]. PVS ist ein allgemeiner Theorembeweiser für Logik höherer Ordnung [GM93]. Die Eingabe von PVS sind Textdateien mit Funktionsdefinitionen[14] und Theoremen. Ist der Quelltext eingelesen, kann man in einem interaktiven Beweisermodus die Theoreme beweisen. Dabei wählt man nacheinander verschiedene Beweiserkommandos, die das aktuelle Beweisziel modifizieren, bis es schließlich einem Axiom entspricht. Die Beweiskommandos gestatten natürlich nicht die Ableitung falscher Sachen.[15]

Im VFisaco-Projekt haben wir bereits einen Teil der im vorigen Abschnitt skizzierten

[14] Eine Logik höherer Ordnung enthält immer auch eine reine funktionale Programmiersprache, so ähnlich wie Haskell [Has, HPF92], in der Funktionen definiert werden können.

[15] Allerdings gibt es ab und zu auch Fehler in PVS mit denen man dann zum Beispiel $1 = 2$ beweisen kann.

```
duff(source, dest : posnat, count : nat) : [Result[State, Unit] -> Result[State,Unit]] =
  write_int(rounds, div(const_int(count), const_int(8))) ##
  write_int(i, const_int(0)) ##
  int_switch_stm(
    rem( const_int(count), const_int(8)),
      int_case(0) ##
              gwhile_stm(
                 const_int(0) < post_decr_const(rounds),
                      skip_res ##
                      write_int_array(dest, read_int(i),
                              read_int_array(source, read_int(i))) ##
                          write_int(i, read_int(i) ++ const_int(1)) ##
      int_case(7) ##  write_int_array(dest, read_int(i),
                              read_int_array(source, read_int(i))) ##
                          write_int(i, read_int(i) ++ const_int(1)) ##
              ..........................................
      int_case(1) ##  write_int_array(dest, read_int(i),
                              read_int_array(source, read_int(i))) ##
                          write_int(i, read_int(i) ++ const_int(1))

              )  % end gwhile_stm
  ) % end int_switch_stm
```

Abbildung 2: Die Semantik von Duff's Device in PVS

Semantik von C++ in PVS formalisiert. In PVS liegt die Semantik in Form von Definitionen für jede mögliche Anweisung vor. Mit Hilfe dieser Definitionen kann man dann die Semantik eines Programmes zusammensetzen. Ein Compiler, der C++ Programme automatisch in ihre Semantik in PVS übersetzen kann ist in Arbeit, aber noch nicht einsatzbereit. Derweil übersetzen wie die zu verifizierenden Programmbeispiele per Hand.

Um Ihnen einen Eindruck zu ermöglichen enthält Abbildung 2 die Definition der Funktion duff — der Semantik von Duff's Device. Abgesehen von den Parametern source, dest und count ist duff eine Funktion so wie im vorherigen Abschnitt besprochen, nur dass die PVS Formalisierung von *Res* eben Result[State, Unit] heißt. Die Definition von duff besteht aus der Verknüpfung der Semantiken für die einzelnen Anweisungen. Dabei ist ## ein PVS infix-Operator für die Komposition von Funktionen.

Mit dem Beweiser von PVS kann man nun Eigenschaften von duff und damit von Duff's Device zeigen. Zum Beispiel auch die von uns entwickelte Spezifikation, deren PVS Quelltext Sie in Abbildung 3 sehen können. Diese Spezifikation beschreibt die geforderten Eigenschaften in der Form eines Theorems in der Logik von PVS. Das Theorem heißt duff_total und besagt, dass die Gleichung ab Zeile 3 unter der Vorbedingung duff_var_ok(...) gilt. Diese Vorbedingung enthält Typkorrektheitseigenschaften. Zum Beispiel, dass die Felder source und dest nur ganze Zahlen enthalten (und nicht etwa Booleans), dass die Variable rounds im Speicher an einer anderen Stelle liegt, als das Feld dest, aber auch, dass beide Felder wenigstens count Elemente enthalten. Abgesehen von dieser letzten Bedingung gelten diese Typkorrektheitseigenschaften natürlich für das Originalprogramm. In der Semantik werden

```
duff_total : Theorem Forall(source, dest : posnat, count : nat) :                    1
    duff_var_ok(source, dest, count, s) Implies                                      2
      duff(source, dest, count)(ok(s,unit)) =                                        3
      ok(s WITH [                                                                    4
          `vars := Lambda(j : posnat) :                                              5
             IF j = rounds Then int(−1)                                              6
             Elsif j = i Then int(count)                                             7
             Elsif cell_in_array(s, dest)(j) And                                     8
                     index_from_cell(s, dest, j) < count                             9
                Then                                                                 10
                   s`vars(get_array(s`vars(source))                                  11
                            `fields(index_from_cell(s, dest, j)))                    12
             Else s`vars(j)                                                          13
             Endif                                                                   14
        ], unit)                                                                     15
```

Abbildung 3: Spezifikation von Duff's Device

sie aber nicht als wahr vorausgesetzt.[16]

Die Gleichung in der Spezifikation besagt, dass die Funktion duff, in einem Zustand s gestartet, immer normal terminiert (dass heißt, die `while`-Schleife terminiert und es gibt unter den gegebenen Vorbedingungen keine Zugriffe außerhalb der Feldgrenzen von source und dest). Desweiteren ist der Zustand, nachdem Duff's Device terminiert hat, genau beschrieben, nämlich als s **WITH** [...]. Dabei enthält der Ausdruck in den eckigen Klammern die Änderungen gegenüber dem Anfangszustand s. Im Endzustand enhält die Variable rounds den Wert -1, i den Wert des Parameters count und die Feldelemente mit Index kleiner als count sind von source nach dest kopiert worden. Mehr wird nicht verändert.

Die gesamten PVS Quellen, einschließlich der Beweise, stehen im WWW unter der Adresse http://wwwtcs.inf.tu-dresden.de/∼tews/Goto/. Der Korrektheitsbeweis für Duff's Device besteht außer der Spezifikation aus weiteren 37 Lemmata, die mit insgesamt ungefähr 700 PVS-Beweiskommandos bewiesen wurden. Die Spezifikation und der Korrektheitsbeweis für Duff's Device sind relativ einfach. Die wirkliche Herausforderung bei Duff's Device lag darin, eine denotationelle Semantik zu entwickeln, die es einem erlaubt, per `switch` in eine Schleife hineinzuspringen.

5 Was wir noch loswerden wollten (Zusammenfassung)

Dieser Artikel gibt einen kurzen Überblick über verschiedene Methoden, zur Verbesserung der Qualität von Software. Der Schwerpunkt liegt dabei auf der Verifikation der Software mit Hilfe einer denotationellen Semantik. Diesen Ansatz verfolgen wir auch im VFiasco-Projekt an der TU Dresden, das sich zum Ziel stellt, einige Eigenschaften des Mikrokernbetriebssystems Fiasco zu verifizieren. Abschnitt 3 beschreibt eines der letzten Forschungsergebnisse

[16] Setzt man Typkorrektheit voraus, kann man Programme, die eine eigene Speicherverwaltung haben nicht mehr behandeln. Denn für solche Programme muss ja erst bewiesen werden, dass die Speicherverwaltung korrekt arbeitet und jeder dynamisch allozierten Variablen ihren eigenen Speicher zuweist.

des Projektes: Eine denotationelle Semantik für C++, die nicht nur abrupte Termination durch `break`, `continue` oder `return` sondern auch Sprungbefehle (`goto`) und berechnete Sprünge (`switch`) korrekt behandelt. Im letzten Abschnitt demonstrieren wir die Anwendbarkeit dieser Semantik durch die Verifikation von Duff's Device.

Literatur

[Coq] *The coq proof assistant.* http://coq.inria.fr/.

[Duf04] Duff, T.: *Tom Duff on Duff's Device*, 2004. available at www.lysator.liu.se/c/duffs-device.html.

[ECCH00] Engler, D., B. Chelf, A. Chou, and S. Hallem: *Checking system rules using system-specific, programmer-written compiler extensions.* In *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 23–25 October 2000.

[GM93] Gordon, M. J. C. and T. F. Melham: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[Hal] *Halteproblem.* Artikel in Wikipedia, der feien Enzyklopädie. Siehe `de.wikipedia.org/wiki/Halteproblem`.

[Has] *The haskell home page.* http://www.haskell.org/.

[HBB+98] Härtig, H., R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter: *DROPS: OS support for distributed multimedia applications.* In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[HH01] Hohmuth, M. and H. Härtig: *Pragmatic nonblocking synchronization for real-time systems.* In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter: *The performance of µ-Kernel-based systems.* In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 66–77, New York, October 5–8 1997. ACM Press.

[HJ00] Huisman, M. and B. Jacobs: *Java program verification via a Hoare logic with abrupt termination.* In Maibaum, T. (editor): *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, Berlin, 2000.

[Hol03] Holzmann, G. J.: *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 1003.

[HPF92] Hudak, P., J. Peterson, and J. H. Fasel: *A Gentle Introduction to Haskell 98.* Available via haskell.cs.yale.edu/tutorial/, October 1992.

[HT03] Hohmuth, M. and H. Tews: *The semantics of C++ data types: Towards verifying low-level system components.* In Basin, D. and B. Wolff (editors): *TPHOLs 2003, Emerging Trends Proceedings*, pages 127–144. 2003. Technical Report No. 187 Institut für Informatik Universität Freiburg.

[JC00] J. Copeland, J. S. Haemer: *The art of software testing.* Server/Workstation Expert, aug 2000. online Journal, `swexpert.com/C9/SE.C9.AUG.00.pdf`.

[lon] *longjmp, siglongjmp — non-local jump to a saved stack context.* Linux manual page.

[LOO] *The loop project.* http://www.cs.ru.nl/sos/research/loop/.

[McM93] McMillan, K.: *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Mod] *Modula-2 record types.* http://www.modula2.org/reference/recordtypes.php.

[Mos90] Mosses, P. D.: *Denotational semantics.* In Leeuwen, J. van (editor): *Handbook of Theoretical Computer Science, volume B*, chapter 11, pages 575–631. Elsevier Science Publishers, Amsterdam, 1990.

[Mye01] Myers, G. J.: *Methodisches Testen von Programmen.* Oldenbourg, 7 Auflage, 2001. Übersetzung von *The art of software testing.*

[ORR+96] Owre, S., S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas: *PVS: Combining specification, proof checking, and model checking.* In Alur, R. and T.A. Henzinger (editors): *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, Berlin, 1996.

[PVS] *Pvs specification and verification system.* http://pvs.csl.sri.com/.

[Tew00] Tews, H.: *Verifying duff's device: A simple compositional denotational semantics for goto and computed jumps.* Submitted, 2000. Available from `wwwtcs.inf.tu-dresden.de/~tews/science.html`.

[Whe02] Wheeler, D. A.: *More than a gigabuck: Estimating gnu/linux's size*, July 2002. Published at `http://www.dwheeler.com/sloc`.