

Certifying Algorithms

Prinzip und einfache Beispiele

Christian Achenbach

FH Wedel

Inhaltsverzeichnis

Certifying Algorithms	1
<i>Christian Achenbach</i>	
1 Warnende Beispiele	3
1.1 Ariane 5	3
1.2 Pentium-FDIV-Bug	3
1.3 Planaritätstest in LEDA 2.0	4
2 Geschichte und Alternativen	4
2.1 Softwaretests	4
2.2 Assertions	4
2.3 Hoare-Kalkül	5
2.4 Model Checking	5
3 Allgemein bekannte zertifizierende Algorithmen	6
3.1 Erweiterter Euklidischer Algorithmus	6
3.2 Neunerprobe	6
4 Definition und formales Framework	7
4.1 Strongly Certifying Algorithms	7
Bipartiter Graph	8
Planarer Graph in LEDA	9
4.2 Certifying Algorithms	10
4.3 Weakly Certifying Algorithms	11
naiver randomisierter SAT-solver	11
5 Vorteile	11
Belegte Korrektheit	11
Tests über alle Eingaben	11
Fehler Identifizierung	11
Vertrauen ohne großen Aufwand	11
Entfernte Berechnung	12
6 Fazit	12

1 Warnende Beispiele

1.1 Ariane 5

Am 4. Juni 1996 wurde der Prototyp der Ariane-5-Rakete der Europäischen Raumfahrtbehörde eine Minute nach dem Start in vier Kilometern Höhe gesprengt. Bei der Entwicklung der Software für die Ariane 5 wurde Spezifikationen der Ariane 4 Rakete übernommen. Durch die wesentlich höhere Beschleunigung kam es bei einer Typkonvertierung einer 64-bit Gleitkommazahl zu einer 16-bit Ganzzahl mit Vorzeichen zu einem arithmetischen Überlauf, welche eine Kaskade von Fehlern nachsichzog und so letztendlich zur Selbstzerstörung der Rakete führte. Vor dem Flug wurden keine Simulationen unter den neuen Bedingungen der Ariane 5 durchgeführt, so konnte der Fehler im Vorfeld nicht entdeckt werden. Der Schaden betrug etwa 370 Millionen US-Dollar.



Abbildung 1. Ariane 5 am 4. Juni 1996

1.2 Pentium-FDIV-Bug

Der 1994 als „Pentium-Bug“ bekannt gewordene Fehler im CPU-Design des kalifornischen Prozessor-Herstellers Intel zeigt, dass selbst fehlerfreie Software eine falsche Ausgabe liefern kann, sofern die darunter liegende Hardware nicht fehlerfrei arbeitet.

Ein Beispiel, welches in beliebiger Software (BASIC, Microsoft Excel, C...) programmiert und auf einem fehlerhaften Pentium-CPU ausgeführt wird, liefert

folgendes Ergebnis:

$$\frac{4195835.0}{3145727.0} = 1.3338204491362410025 \quad \text{korrekter Wert}$$

$$\frac{4195835.0}{3145727.0} = 1.3337390689020375894 \quad \text{Ergebnis des betroffenen CPUs}$$

1.3 Planaritätstest in LEDA 2.0

Der Planaritätstest in der Algorithmenbibliothek LEDA 2.0 enthielt einen Fehler, so dass manche Graphen falsch klassifiziert wurden. In LEDA 2.1 wurde der Algorithmus korrigiert und durch einen zertifizierenden Algorithmus ersetzt. Zunächst hatte der Algorithmus eine quadratische Laufzeit. Dies wurde aber bald durch den Einsatz eines Algorithmus mit linearer Laufzeit verbessert.

2 Geschichte und Alternativen

2.1 Softwaretests

„Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden.“ [7]

Softwaretests dienen also der Verifikation von Software.

Es werden Aktionen durchgeführt, welche das Programm mit vorher festgelegten Eingaben testet. Um eine Fehlerwirkung feststellen zu können, werden die vom Programm während oder nach der Ausführung gelieferten Ausgaben mit ebenfalls vorher festgelegten erwarteten Ergebnissen verglichen.

Allerdings können Softwaretests nur mit vorher definierten und korrekten Eingabe/Ausgabenpaaren durchgeführt werden.

Siehe auch: [3]

2.2 Assertions

Der Begriff Assertions (Zusicherungen) stammt aus dem 1967 veröffentlichten Artikel „Assigning Meanings to Programs“ von Robert Floyd [4].

Die Wirkung eines Programms durch Verarbeitungsschritte kann als Transformation von Eingabe in Ausgabe betrachtet werden. Die Situation vor dem Verarbeitungsschritt wird aus der Situation danach abgeleitet. Solche Situationen werden in Form von sogenannten Zusicherungen (Assertions) angegeben.

Ein Beispiel soll dies in Java verdeutlichen:

```
int A = 5, B = 10;
int x = A, y = B;

assert x==A & y==B;

while (x > 0)
{
    y = y+1;
    x = x-1;
}

assert y==A+B;
```

Allerdings kann man leicht einsehen, dass dieses Beispiel nicht mehr funktioniert sobald sich der Wert für A im negativen Bereich befindet. In diesem Fall würde die zweite Zusicherung einen Fehler identifizieren. Die Schwäche der Assertions wird deutlich: Zusicherungen können zwar getestet, aber nicht verifiziert werden.

Siehe auch: [1] und [7]

2.3 Hoare-Kalkül

Der Hoare-Kalkül [5] dient zur Verifikation der partiellen und totalen Korrektheit (kleiner!) Programme. Grundlage ist folgende Überlegung: Ein Programm, welches syntaktisch korrekt in einer bestimmten Programmiersprache formuliert wurde, enthält alle Informationen um die Folgen einer Programmausführung zu ermitteln. Das Verhalten eines Programms wird dann durch Anwendung von Regeln mit Hilfe einiger gültiger Axiome ermittelt. Diese Axiome beschreiben die Eigenschaften der Elemente der Programmiersprache. Damit entsteht eine maschinenunabhängige Beschreibung. Mit dieser dann eine Überprüfung der Korrektheit des Programms auf dem Quellcodeniveau erfolgt.

2.4 Model Checking

Model Checking beschreibt ein Verfahren zur automatischen Verifikation von Software und reaktiven Systemen. Vorgestellt wurde diese Technik erstmals 1981 von Clarke und Emerson. [2]

Ziel ist es auf einem Modell eines Software- oder Hardwaresystems zu prüfen, ob das Modell die zu verifizierende Systemeigenschaft besitzt. Im Gegensatz zu Tests und Simulation werden hierbei alle möglichen Verhalten eines Systems untersucht. Allerdings bedarf es dafür einen recht hohen Aufwand für die Modellierung und Spezifikation.

3 Allgemein bekannte zertifizierende Algorithmen

3.1 Erweiterter Euklidischer Algorithmus

Der erweiterte euklidische Algorithmus berechnet neben dem größten gemeinsamen Teiler ($\text{ggT}(a,b)$ mit $a,b \in \mathbb{N}$) auch noch zwei ganze Zahlen s und t :

$$(\text{ggT})(a,b) = s * a + t * b$$

Eine rekursive Implementierung ist hier beispielhaft gezeigt:

```

extended_euclid(a,b)
1  wenn b = 0
2      dann return (a,1,0)
3  (d',s',t') <-- extended_euclid(b, a mod b)
4  (d,s,t) <-- (d',t',s' - floor(a/b)t')
5  return (d,s,t)

```

Ein Beispiel zeigt das Ergebnis: $(\text{ggT})(99,78) = 3 = -11 * 99 + 14 * 7 = 3$

Offensichtlich erzeugt diese Berechnung durch ihren Algorithmus eine Art Zeugen, hier s und t , durch die das Ergebnis schnell und unkompliziert überprüft werden kann.

3.2 Neunerprobe

Schon bereits vor über 1000 Jahren wurde von Al-Kharizmi ein Verfahren zur (teilweisen) Erkennung von Rechenfehler in einer Multiplikation beschrieben. Zum Beispiel soll geprüft werden, ob $c = a * b$. Ganz im Sinne der zertifizierenden Algorithmen lässt sich die Korrektheit einer längeren Rechnung anhand einer leichten Nebenrechnung überprüfen. Von allen an einer Rechenoperation beteiligten Zahlen werden die Neunerreste ermittelt. Dafür wird solange die Quersumme einer Zahl berechnet, bis nur noch eine einstellige Zahl übrigbleibt.

Ein Beispiel:

$$5919 : 5 + 9 + 1 + 9 = 24 : 2 + 4 = 6$$

Danach wird der Rechenweg mit den Neunerresten der Zahlen durchgeführt. Falls das Ergebnis der zweiten Rechnung nun mit dem Neunerrest des Ergebnis der ersten Rechnung übereinstimmt, wurde die Rechnung mit hoher Wahrscheinlichkeit korrekt durchgeführt.

Beispiel:

$$573 * 492 = 281916 \rightarrow 6 * 6 = 9 \rightarrow 36 = 9 \rightarrow 9 = 9$$

4 Definition und formales Framework

Im folgenden Abschnitt werden Algorithmen mit Eingaben aus der Menge X und Ausgaben aus der Menge Y betrachtet. Für die Eingaben können Vorbedingungen $\varphi(x)$, bzw. für die Ausgaben Nachbedingungen $\psi(x, y)$ formuliert werden. Das Paar (φ, ψ) wird I/O-Spezifikation genannt.

Zusammengefasst:

1. Eingabe: $x \in X$
2. Ausgabe: $y \in Y$
3. Vorbedingung: $\varphi : X \rightarrow \{T, F\}$
4. Nachbedingung: $\psi : X \times Y \rightarrow \{T, F\}$
5. I/O-Spezifikation: (φ, ψ)

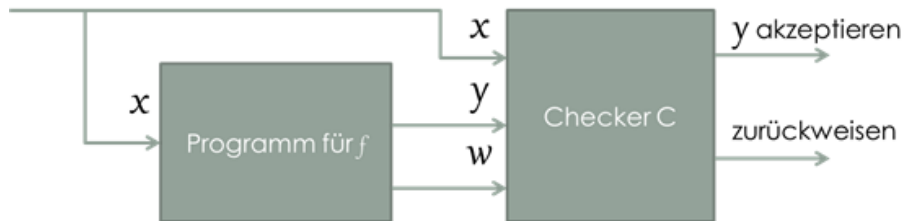


Abbildung 2. Schematische Darstellung des Eingabe/Ausgabe-Verhaltens eines zertifizierenden Alorithmus.

4.1 Strongly Certifying Algorithms

Die Klasse der stark zertifizierenden Algorithmen ist die wünschenswerteste Klasse. Sie zeigt dem Benutzer für jede Eingabe die Korrektheit des Algorithmus, oder das eine illegale Eingabe dem Algorithmus vorgegeben wurde. Sofern die Vorbedingung nicht verletzt wurde gibt der stark zertifizierende Algorithmus neben der Ausgabe y auch einen Zeuge w mit aus. Andernfalls wird eine Verletzung der Vorbedingung (hier durch \perp dargestellt) angezeigt.

1. Erweiterte Ausgabe: $Y^\perp := Y \cup \{\perp\}$
2. \perp = Indikator für verletzte Vorbedingung
3. W = Die Menge der Zeugen (witnesses)
4. (φ, ψ) hat die Struktur $W : X \times Y^\perp \times W \rightarrow \{T, F\}$

So gilt demnach: $\forall x, y, w$
 $(y = \perp \wedge W(x, y, w)) \Rightarrow \neg\varphi(x)$ und
 $(y \in Y \wedge W(x, y, w)) \Rightarrow \psi(x, y)$

Falls $y = \perp$: Dann beweist w , dass x nicht die Vorbedingung erfüllt. Falls $y \in Y$: Dann beweist w , dass (x, y) die Nachbedingung erfüllt.

Bipartiter Graph

Definition: „Ein einfacher Graph $G = (V, E)$ (V Menge der Knoten, E Menge der Kanten) heißt bipartit oder paar, falls sich seine Knoten in zwei disjunkte Teilmengen A und B aufteilen lassen, sodass zwischen den Knoten innerhalb beider Teilmengen keine Kanten verlaufen.“ [8]

Eine zweite Definition lässt eher erkennen wie ein zertifizierender Algorithmus arbeiten könnte:

Definition: „Ein Graph $G = (V, E)$ wird bipartit genannt, wenn seine Knoten zweifarbig (schwarz, gelb) markiert werden können, so dass jede Kante des Graphen einen schwarzen und einen gelben Knoten verbindet“ [6]

Ein nicht bipartiter Graph zeichnet sich durch die Existenz eines ungeraden Kreises aus. Dieser ungerade Kreis besteht aus mehrere Knoten, welche durch ihre Kanten in einem Kreis verbunden sind $(v_0, v_1), (v_1, v_2), \dots, (v_{2k}, v_{2k+1})$ und $v_0 = v_{2k+1}$. Alle Knoten v_i und v_{i+1} haben nun unterschiedliche Farben. Aber $v_0 = v_{2k+1}$ and damit v_0 müsste beide Farben haben, dies führt zu einem Widerspruch.

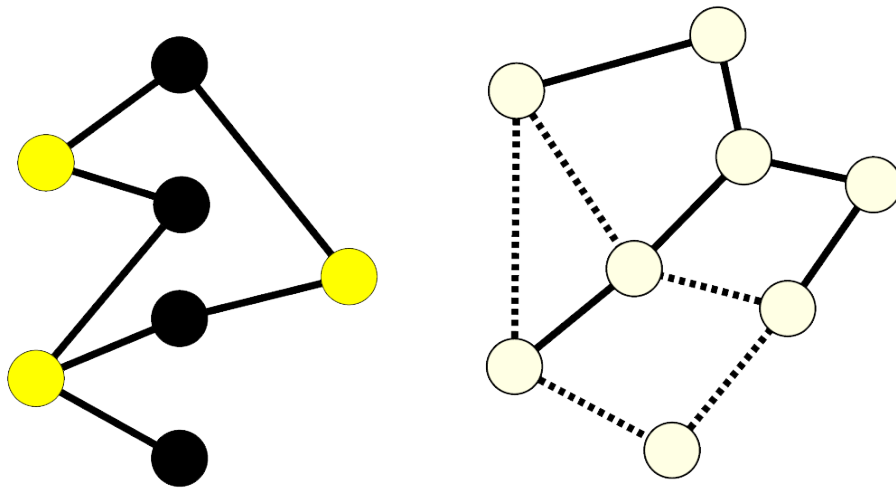


Abbildung 3. Links: Bipartiter Graph, gezeigt durch die Einfärbung der Knoten mit zwei Farben. Rechts: Nicht bipartiter Graph, gezeigt durch den Kreis mit ungerader Länge. Entnommen aus [6]

Planarer Graph in LEDA

Definition: Ein Graph heisst planar, wenn man ihn so zeichnen kann, dass sich keine Kanten kreuzen. Theorem: (Kuratowski) Ein Graph G ist genau dann planar, wenn G keinen Teilgraphen G' enthält, so dass K_5 oder $K_{3,3}$ ein Minor von G' ist.

Die Algorithmenbibliothek LEDA zeigt eindrucksvoll zwei Zeugen ihres zertifizierenden Algorithmus für bipartite Graphen:

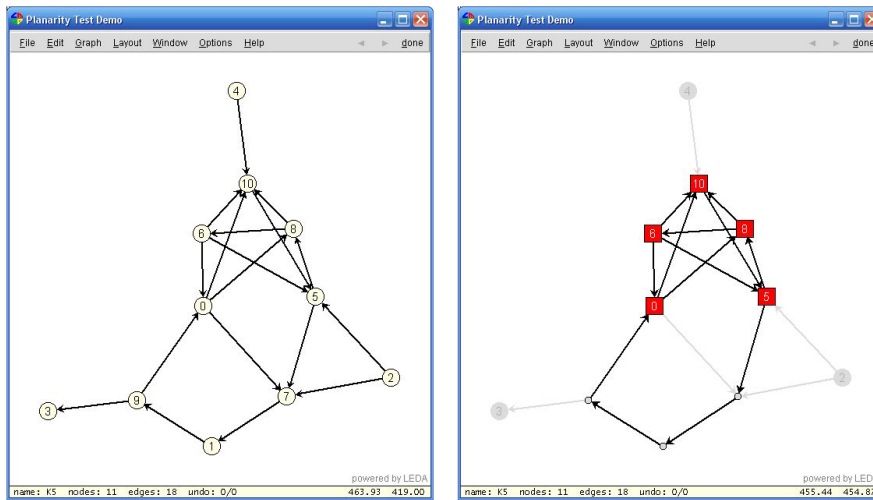


Abbildung 4. Zeuge für einen nicht planaren Graph in LEDA. Zeuge: Gezeigter K_5

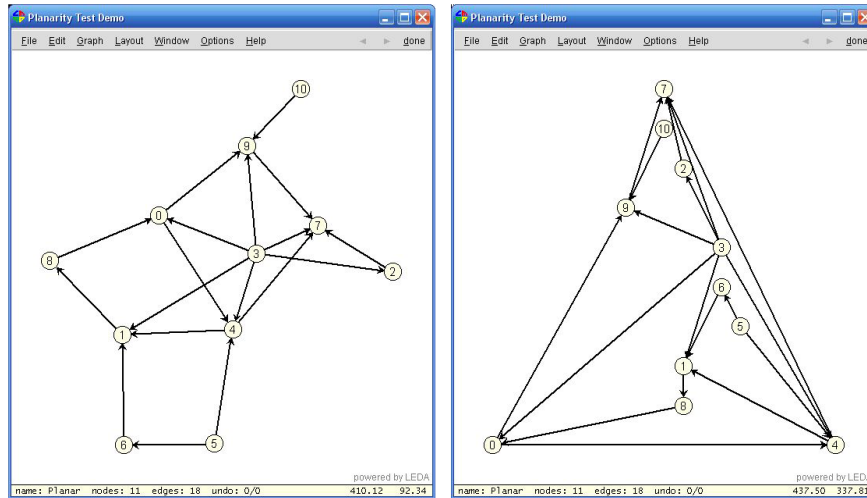


Abbildung 5. Zeuge für einen planaren Graph in LEDA. Zeuge: Seine planare Darstellung

4.2 Certifying Algorithms

In manchen Fällen kann ein stark zertifizierender Algorithmus nicht erreicht werden. Diese Art von zertifizierenden Algorithmen kann beweisen, dass entweder die Vorbedingung oder die Nachbedingung nicht erfüllt wurde. Allerdings kann nicht gezeigt werden welche von beiden Bedingungen verletzt wurde.

Es gilt also: $\forall x, y, w$
 $y = \perp \wedge W(x, y, w) \Rightarrow \neg\varphi(x)$ und
 $y = \in Y \wedge W(x, y, w) \Rightarrow \neg\varphi(x) \vee \psi(x, y)$

Ein Beispiel wäre die binäre Suche. Als Eingabe wird dem Algorithmus ein Array mit Zahlen übergeben. Nun gibt es zwei Möglichkeiten:

1. Das Array ist korrekt sortiert: In diesem Fall wird die gesuchte Zahl gefunden, falls sie existiert. Dies wird durch die Rückgabe des gefundenen Platzes im Array bewiesen. Andernfalls wird die Zahl nicht gefunden.
2. Das Array ist nicht korrekt sortiert: In diesem Fall kann die gesuchte Zahl zwar im Array existieren, aber sie wird unter Umständen nicht gefunden. Eine Verletzung der Vorbedingung kann allerdings durch den Algorithmus auch nicht festgestellt werden.

4.3 Weakly Certifying Algorithms

Schwach zertifizierende Algorithmen terminieren für alle Eingaben $x \in X$, welche die Vorbedingung erfüllen. Falls sie die Vorbedingung nicht erfüllen, dann kann eine Terminierung nicht garantiert werden.

Falls der Algorithmus für eine Eingabe $x \in X$ terminiert, dann erzeugt er eine Ausgabe $y \in Y^\perp$ und einen Zeugen $w \in W$, so dass $W(x,y,z)$ erfüllt ist.

naiver randomisierter SAT-solver

$\varphi(x)$ = (x ist eine erfüllbare boolesche Formel) Der Algorithmus versucht auf naive Weise, eben durch randomisiertes Ausprobieren, eine gültige Belegung für eine boolesche Formel zu finden. Falls eine Belegung gefunden wird, dann terminiert dieser Algorithmus und gibt als Beweis die Belegung der Variablen mit aus. Solange aber keine Belegung gefunden wird terminiert der Algorithmus nicht.

5 Vorteile

Belegte Korrektheit

Falls der Checker C das Tripel (x,y,z) akzeptiert, dann wird durch den Zeugen w bewiesen, dass entweder $\neg\varphi(x)$ oder $\psi(x,y)$ gilt. Mit anderen Worten: Die Vorbedingung wurde verletzt, oder der Algorithmus hat mit der Eingabe x die korrekte Ausgabe y geliefert.

Falls das Tripel (x,y,z) allerdings abgelehnt wird, dann können wir das ausgehen, dass entweder die Ausgabe y oder der Zeuge w falsch berechnet wurden.

Tests über alle Eingaben

In herkömmlichen Testverfahren (Unit-Tests) müssen zunächst korrekte Eingabe/Ausgabepaare erdacht werden. Dies ist zeit,- und fehlerabfällig. Durch zertifizierende Algorithmen können Tests über alle Eingabe ausgeführt werden. Sogar im produktiven Einsatz werden die Algorithmen noch „gestet“.

Fehler Identifizierung

In großen Programm können Fehlerquellen häufig nur sehr schwer identifiziert werden. Folgefehler breiten sich schnell im System aus und werden erst im fortgeschrittenen Programm ablauf sichtbar. Durch zertifizierende Algorithmen können Fehlerquellen sofort identifiziert, und die weitere Ausbreitung des Fehler im System verhindert werden.

Vertrauen ohne großen Aufwand

In einigen Fällen ist eine Verifizierung der Korrektheit eines komplizierten Algorithmus nur sehr schwer oder garnicht erreichbar. Sobald aber ein einfach zu verstehender Checker C, dessen Korrektheit schnell verifiziert werden kann, die Aushaben des komplizierten Algorithmus überprüft, steigt das Vertrauen in das Gesamtsystem.

Hinzukommt, dass ein Checker auch in einer anderen Sprache implementiert werden kann als der eigentlich Algorithmus. Diese Sprache muss nicht unbedingt allen Anforderungen genügen welche an den Algorithmus gestellt werden.

Entfernte Berechnung

Berechnung welche von einer nicht vertrauenswürdigen, oder stör anfälligen Quelle geliefert werden können nachträglich lokal durch einen Checker überprüft werden.

6 Fazit

Certifying algorithms are a preferred kind of algorithm. They prove their work and they are easier to implement reliably. Their wide-spread use would greatly enhance the reliability of algorithmic software. [6]

Literatur

1. M. BROY, *Informatik: eine grundlegende Einführung*, Springer, 1998.
2. E. C. CLARKE, O. GRUMBERG, AND D. A. PELED, *Model Checking*, MIT Press, 2000.
3. E. DENERT, *Software-Engineering*, Springer Berlin, 1992.
4. R. W. FLOYD, *Assigning meanings to programs*, in Proceedings of Symposia in Applied Mathematics, vol. 19, 1967, pp. 19–32.
5. C. A. R. HOARE, *An axiomatic basis for computer programming*, in Comm. of the ACM, vol. 12(10):576-585, October 1969.
6. R. MCCONNELL, K. MEHLHORN, S. NÄHER, AND P. SCHWEITZER, *Certifying algorithms*, Computer Science Review, (June 2010).
7. G. J. MYERS, *Methodisches Testen von Programmen*, Oldenbourg, München, Wien, 1991.
8. UNBEKANNT, *Bipartiter graph*. Wikipedia.