1) Einführung in die formale Verifikation

Wofür formale Verifikation?

steigender Einsatz von Computertechnologie im Alltag

- Internet, Mobiltelefonie, E-Banking, Flugzeug- und Autosteuerungen, Börsentransaktionen, medizinische Systeme ...
- durchschnittliche Anzahl Benutzungen pro Mensch und Tag 1995: 25

Versagen solcher Systeme ist im Bereich zwischen

- ärgerlich
- ...
- lebensbedrohend

Was so alles passieren kann . . .

. . . wenn die Programme nicht korrekt funktionieren

- hoher ökonomischer Schaden für Hersteller oder Anwender
- herber Verlust
- unnötige Maintenance
- . . .

Beispiele:

- Intel™ Pentium™ II FPU Bug 1994
  - $420.000.000 von Intel zur Schadensbehebung bereitgestellt
  - einige Einträge in Tabelle für Divisionen fehlten

Was so alles passieren kann . . .

. . . wenn die Programme nicht korrekt funktionieren

- Softwarefehler im Gepäcktransportsystem des Flughafens in Denver 1993-94
  - $1.100.000 Verlust pro Tag über 9 Monate hinweg
  - ganzes System instabil
- Ausfall des Kommunikationssystems am Flughafen L.A. 2004
  - plötzlich kein Kontakt zu ca. 400 Flugzeugen in der Luft
  - System nutzte 32Bit-Integer für Countdown-Timer in msec
  - 2^{32} msec $\approx$ 50 Tage $\rightsquigarrow$ automatischer Shutdown nach 7 Wochen
  - Abhilfe: Techniker rebootet alle 30 Tage
- Crash der Ariane-5-Rakete 1997
  - fehlgeschlagene Konvertierung von 64Bit- auf 16Bit-Wert mit Shutdown
  - Backup-System war identisch

Was so alles passieren kann . . .

. . . wenn die Programme nicht korrekt funktionieren

- Mars Pathfinder 1997
  - ständige Resets mit jeweiligen Datenverlust
  - Bug in einem Task Scheduler: niedrig priorisierter Thread konnte höher priorisierten Thread blockieren
- Therac-25 Bestrahlungsmaschine 1985-1987
  - automatische Steuerung ließ 100fache Überdosierung zu
  - zwei verschiedene Modi (Intensität und Ausrichtung) vermischt
  - 3 von 6 Krebspatienten kurz danach tot
- . . .

mehr auf

`http://www.cse.lehigh.edu/~gtan/bug/softwarebug.html`

Warum sind Systeme fehleranfällig?

verschiedene Gründe:

- Größe

  Windows XP ca. 40.000.000 Zeilen Sourcecode

- Modularisierung

  - Komponenten interagieren über Schnittstellen, die spezifiziert werden müssen
  - Umgebung einer Komponente ist evtl. unbekannt
  - Zustandsraum ist exponentiell in Anzahl involvierter Komponenten

- Nebenläufigkeit

  race conditions bei Multi-Thread-Systemen schwer vorhersehbar

- ...

Verifikation

Verifikation = Erbringen eines Beweises dass ein gegebenes Programm sich an eine gegebene Spezifikation hält

vage: das Programm ist *korrekt*

Programm ist hier erst einmal abstrakt zu verstehen: Hardware, Software, etc.

beachte: Verifikation ist im Allgemeinen natürlich unentscheidbar (warum?)

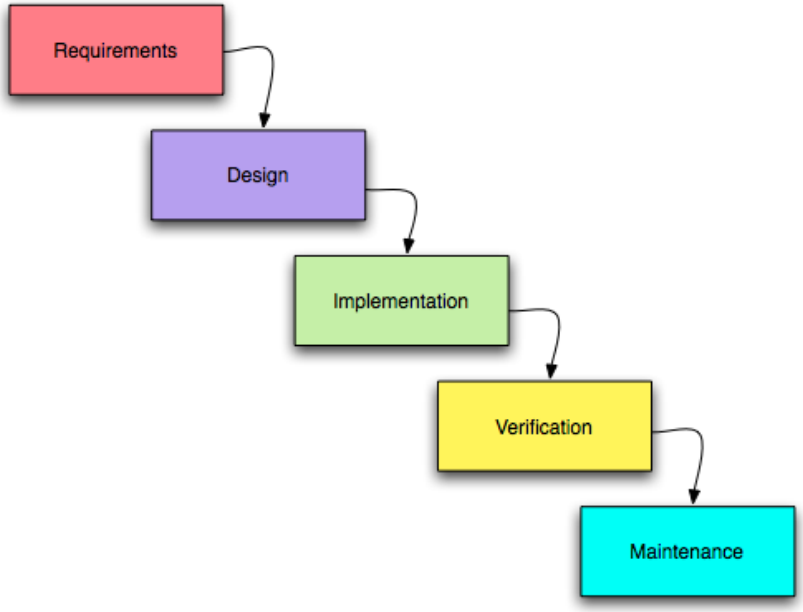
verschiedene Techniken zur Verifikation gewisser Programme und gewisser Eigenschaften

Verifikation mittlerweile üblicher und essentieller Bestandteil der Systementwicklung

Der Designzyklus

Programme, insbesondere große Systeme, werden nicht ad-hoc erstellt

siehe z.B. Wasserfallmodell aus Software Engineering



beachte: muss nicht so sein!

Verifikation im Designzyklus

wird Fehler während Verifikationsphase entdeckt, so muss man wieder weiter oben einsteigen

je früher Fehler entdeckt werden, desto besser:

- spart Entwicklungszeit und -kosten
- spät entdeckte Fehler machen evtl. Arbeit redundant
- bei Hardware am Ende keine Korrektur mehr möglich

daher Verifikation oft

- früh im Designzyklus
- auf Modellebene statt auf Implementationsebene

Anforderungen an gute Verifikationstechniken

wie immer: schnell, billig, gut; aber was heißt das genau?

- Geschwindigkeit: Designzyklus nicht unnötig verzögern
- Kosten: Designzyklus nicht unnötig verteuern
- Bedienbarkeit: soll von Software-Ingenieuren durchgeführt werden können
- Integrierbarkeit: soll nicht erst am fertigen Produkt durchführbar sein
- Sicherheit: Verifikation selbst soll nicht denselben Fehlerquellen unterliegen wie Programmierung

Anforderungen an gute Verifikationstechniken

- Automatisierbarkeit: wegen Geschwindigkeit und Sicherheit
- Vollständigkeit: nach erfolgreich durchgeführter Verifikation soll sichergestellt sein, dass Programm der Spezifikation genügt
- Gegenbeispiele: soll nicht nur Anwesenheit von Fehler anzeigen, sondern diesen auch lokalisieren
- Spitzfindigkeit: sollte insbesondere auch Fehler finden, die
  - nicht offensichtlich sind
  - nur selten auftreten
- Messbarkeit: sollte Maß an Güte eines Programms nach Verifikation zulassen
- ...

Verifikationstechniken

- Peer Review
  - an der Programmierung Unbeteiligte inspizieren Code
  - Form der manuellen, statischen Analyse
  - findet i.A. viele Fehler, ist jedoch nicht vollständig
  - Fehler durch Nebenläufigkeit oder falsche Algorithmen schwer zu entdecken
- Emulation
  - verwendet in Hardwareverifikation
  - rekonfigurierbarer Chip wird programmiert, so dass er sich wie der gewünschte verhält
  - weiter wie bei Testen
- Simulation
  - wie Emulation, jedoch wird Modell des Chips in Software simuliert

Verifikationstechniken

- Testen
  - Form der dynamischen Analyse
  - Testfälle und jeweilige Ausgabe laut Spezifikation werden vorgegeben und überprüft
  - kann (teilweise) automatisiert werden
  - nicht vollständig: Testen kann Fehler finden, aber nicht deren Abwesenheit aufzeigen
  - komplementiert Peer Review recht gut
  - gute Testfälle zu finden fast so schwer wie Fehler zu finden
  - Problem: Testumgebung
  - Maß für Qualität schlecht an Anzahl der Tests festzulegen
- ...

Der Begriff der Korrektheit

Frage: Ist das folgende Programm korrekt?

```
public static void main(String[] args) {
   int lo = 1;
   int hi = 1;
   System.out.println(lo);
   while (hi < 50) {
      System.out.print(hi);
      hi = lo + hi;
      lo = hi - lo;
   }
}
```

Korrektheit ...

- bezieht sich immer auf eine Spezifikation
- ist keine absolute Eigenschaft eines Programms

Der Begriff der Korrektheit

Frage: Ist das folgende Programm korrekt?

```
public static void main(String[] args) {
   int lo = 1;
   int hi = 1;
   System.out.println(lo);
   while (hi < 50) {
      System.out.print(hi);
      hi = lo + hi;
      lo = hi - lo;
   }
}
```

Korrektheit ...

- bezieht sich immer auf eine Spezifikation
- ist keine absolute Eigenschaft eines Programms

Modelle und Programme

Nächste Frage: Gibt das Programm irgendwann einmal 34 aus?

```
public static void main(String[] args) {
   int lo = 1;
   int hi = 1;
   System.out.println(lo);
   while (hi < 50) {
      System.out.print(hi);
      hi = lo + hi;
      lo = hi - lo;
   }
}
```

hängt natürlich von der Semantik ab!

zur Verifikation müssen Programme als semantische Objekte (nicht nur als syntaktische) angesehen werden

Modelle stellen Semantik von Programmen dar

Modelle

Herkunft:

- abstrakte Ebenen: in spezifischen Sprachen beschriebene Systeme, z.B.
  - Programmflussdiagramme
  - UML (Klassen-, Interaktions-Diagramme, ...)
  - Hardware-Beschreibungssprachen (Verilog, VHDL, ...)
  - Prozess-Beschreibungssprachen (Promela, Petrinetze, Prozessalgebren, ...)
  - ...
- Prototypen: üblicherweise schnell und nur für Testzwecke realisiertes System
- Abstraktion: aus gegebenem Programm wird (üblicherweise vereinfachtes) extrahiert, welches das Verhalten des ursprünglichen nachbildet (siehe z.B. Datenabstraktion)

Schwächen der modellbasierten Verifikation

wichtig: Resultate der Verifikation sind immer nur so gut wie die Modellbildung

Bsp.:

1. Java-Sourcecode wird abstrahiert in Transitionssystem
2. Transitionssystem wird verifiziert
3. verwendeter Java-Compiler ist fehlerhaft

$\rightsquigarrow$ Verifikation sagt nichts über Kompilat aus

Modell-Programm-Beziehung muss treu sein

Formale Verifikation

Verifikation . . .

- auf Ebene mathematische präziser und eindeutiger Modelle
- nach eindeutig festgelegten Regeln durchzuführende Verifikation

Techniken der formalen Verifikation:

1. modellbasierte Simulation
2. modellbasiertes Testen
3. Model Checking
4. Theorembeweisen
5. . . .

Modellbasierte Simulation

benutzt Software-Tool (Simulator), um Systemverhalten in gewissen Szenarien zu untersuchen

Szenarien sind benutzerdefiniert oder automatisch generiert

- \+ leicht durchzuführen
- \+ üblicherweise schnell
- \+ liefert Gegenbeispiele
- – nicht geeignet um subtile Fehler aufzuspüren
- – nicht geeignet um Maß an Güte des Systems festzulegen
- – kann Abwesenheit von Fehlern nicht belegen

etc.

Modellbasiertes Testen

Testfälle werden automatisch aus Modell des Systems gewonnen

- \+ leicht durchzuführen
- \+ üblicherweise schnell
- \+ liefert Gegenbeispiele
- – nicht geeignet um subtile Fehler aufzuspüren
- – nicht geeignet um Maß an Güte des Systems festzulegen
- – kann Abwesenheit von Fehlern nicht belegen

etc.

Theorembeweisen

durch logische Formeln dargestellt:

- zu verifizierende Korrektheitseigenschaft $\Phi_{correct}$
- Programmverhalten Φ_{model}

Verifikation: System ist korrekt gdw. $\models \Phi_{model} \Rightarrow \Phi_{correct}$

Theorembeweiser = Tool zum (interaktiven) Führen von Beweisen in jeweiliger Logik

oft eingebaute Programmiersprache, um direkt über Programme reden zu können

Theorembeweisen

beachte: Prädikatenlogik FO bereits unentscheidbar

daher Theorembeweisen oft "nur" Theorem-Assistent

- − langsam
- − sehr arbeitsintensiv, falls Taktiken nicht anschlagen
- − fehleranfällig wegen Hauptlast auf Benutzer
- − oft nicht erfolgreich
- − erfordert u.U. Expertise auf Gebiet des Theorembeweisens
- \+ kann Abwesenheit von Fehlern aufzeigen
- \+ kann Aussagen über Programmen mit unendlichem Zustandsraum beweisen

Existierende Theorembeweiser

- Coq:
  - Logik: Kalkül induktiver Definitionen
  - Programme werden aus Beweisen extrahiert
  - `http://www.lix.polytechnique.fr/coq/`
- Isabelle:
  - Logiken: FO, HOL, ZF
  - Programme: ML
  - `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`
- PVS:
  - Logik: HOL
  - `http://pvs.csl.sri.com/`
- ...

Model Checking

Modell wird als mathematische Struktur $\mathfrak{A}_{model}$ angesehen

Korrektheitseigenschaft wird als logische Formel $\Phi_{correct}$ formuliert

Verifikation = Überprüfen, ob Struktur als logische Interpretation ein Modell der Formel ist

$$\mathfrak{A}_{model} \models \Phi_{correct} \quad ?$$

algorithmisch: Exploration des gesamten Modells in Bezug auf die gegebene Korrektheitseigenschaft

beachte: Begriff *Modell* hier in zweierlei Hinsicht verwendet

- Beschreibung des Verhaltens eines Programms (also aus Modellierung)
- erfüllende Interpretation (aus Logik)

Model Checking durchführen

typischerweise 3 Phasen

1. Modellierungsphase

   Modell des Systems und Formel, die Korrektheitseigenschaft beschreibt, erstellen

2. Laufphase

   Model Checker (Software-Tool) führt Überprüfung durch

3. Analysephase

   falls System die Korrektheitseigenschaft ...

   - erfüllt, dann mache weiter im Designzyklus
   - nicht erfüllt, dann repariere / entwerfe neu / verfeinere Systemmodell / Design / Eigenschaft
     (unter evtl. Zuhilfenahme eines Gegenbeispiels, welches der Model Checker geliefert hat)

Model-Checking: Vor- und Nachteile

– erfordert ein wenig Expertise auf Gebiet der Logik als Spezifikationssprache
(jedoch typischerweise weniger als beim Theorembeweisen)

– Anwendungsbereich limitiert auf “kleine” Modelle
(je nach Technik und Anforderung verschiedene Bedeutungen)

\+ kann die Abwesenheit von Fehlern aufzeigen

\+ kann üblicherweise Gegenbeispiele liefern

etc.

Stärken des Model-Checkings

- anwendbar auf eine Vielfalt von Systemen
- partielle Verifikation: eine Eigenschaft nach der anderen
- Auffindbarkeit von Fehlern hängt nicht von deren Auftrittwahrscheinlichkeit ab
- liefert diagnostische Information
- potentielle Push-Button-Technologie
- in der Industrie immer häufiger eingesetzt
- in existierende Designzyklen integrierbar
- basiert auf mathematischer Theorie

Schwächen des Model-Checkings

- nicht gut geeignet für datenintensive Anwendungen
- limitiert durch Unentscheidbarkeitsresultate
- nur Verifikation eines Modells, nicht des Systems selbst
- nur angesagte Anforderungen werden überprüft
- State-Space-Explosion-Problem
- ganz ohne Expertise auf dem Gebiet geht es schlecht
- nicht geeignet für Generalisierungen (z.B. n-Philosophen statt 5-Philosophen)
- Model-Checker selbst könnte inkorrekt sein

Existierende Model Checker

- NuSMV: symbolischer MC für LTL, CTL, erweiterbar
  `http://nusmv.fbk.eu/`
- Spin: automatenbasierter MC für LTL
  `http://spinroot.com/spin/whatispin.html`
- UPPAAL: MC für Echtzeitsysteme und Echtzeit-CTL
  `http://www.uppaal.com/`
- Java Pathfinder: MC für Java-Programme
  `http://babelfish.arc.nasa.gov/trac/jpf`
- ...

Andere Methoden zur Qualitätssicherung

Model Checking ersetzt nicht andere Methoden

- Software Engineering
- Proof-Carrying Code
- Programmsynthese
- ...