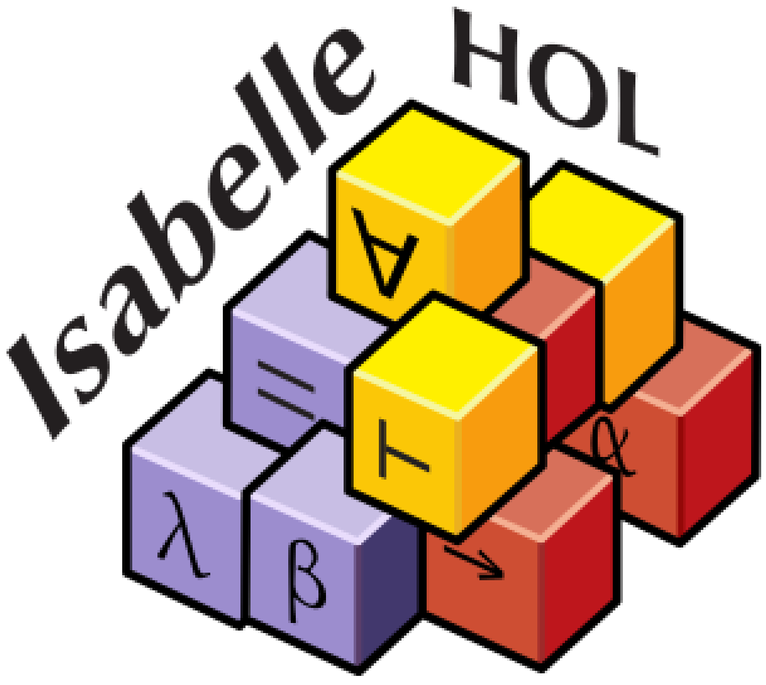
Software korrekt beweisen mit Isabelle

Bianca Lutz

Software Architecture Summit 2023

11. September 2023, Berlin



Isabelle Webseite:

https://isabelle.in.tum.de

Materialien zum Workshop:

https://github.com/sowilo/isabelle-ws-2023

Was wir heute vorhaben

1. Einführung in formale Methoden

   Motivation, Kosten und Nutzen verschiedener Techniken
   Case-Study seL4: Ansätze zur Verifikation komplexer Systeme

2. Praxis-Workshop

   Programmieren und Beweisen in Isabelle/HOL

Formale Methoden

Was sind formale Methoden?

"Der Begriff Formale Methode bezeichnet … eine Vielzahl von … Techniken zur Modellierung und **mathematisch rigorosen** Überprüfung von Computersystemen."

— Wikipedia

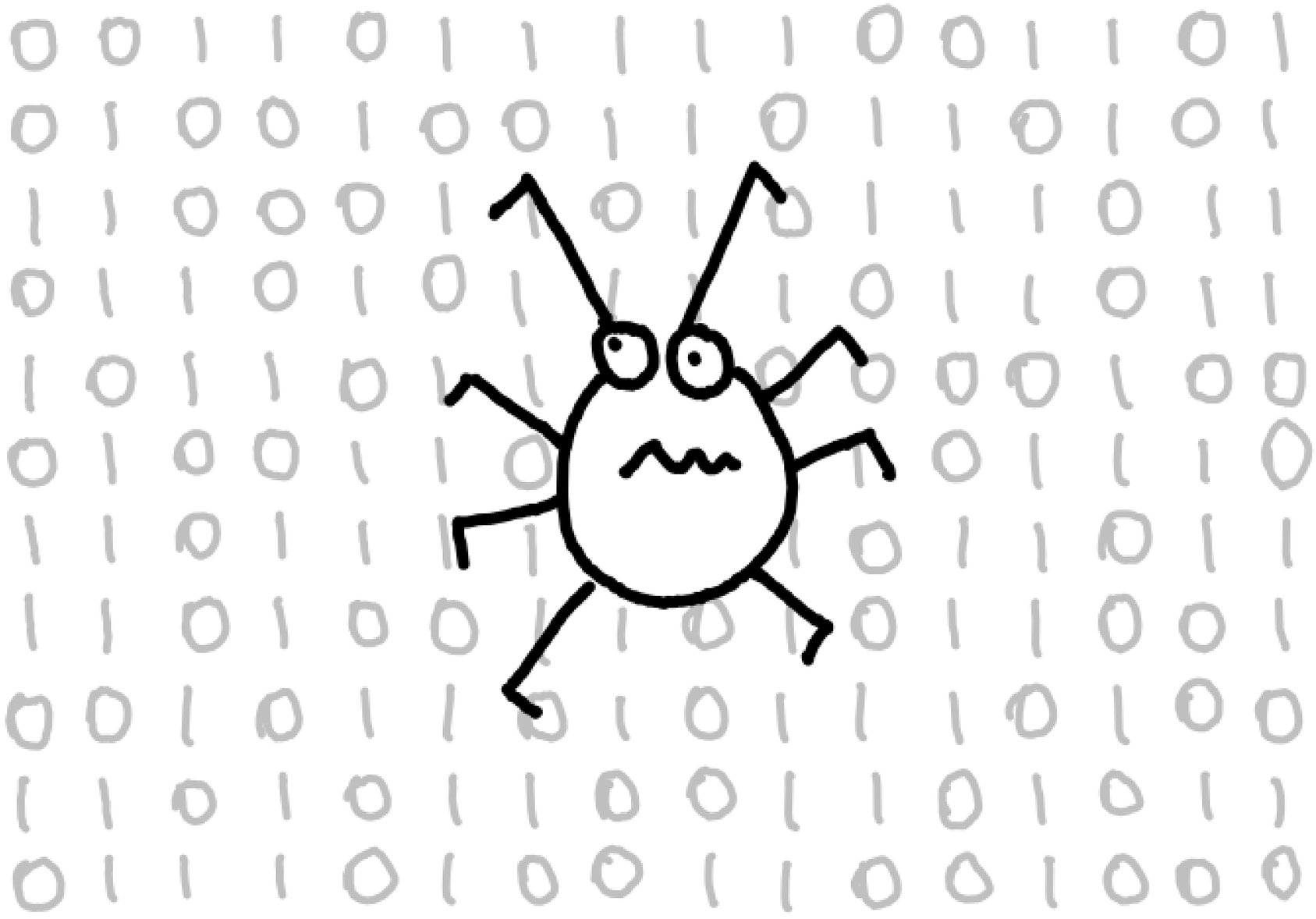
Formale Spezifikation

Verwende mathematische Notation, um das geforderte Systemverhalten präzise zu beschreiben.

Formale Verifikation

Verwende logische (Ableitungs-)Regeln, um mathematisch (rigoros) nachzuweisen, dass das System die Spezifikation erfüllt.

IT-Systeme = Bugs



Wann sind Fehler kritisch?

- Wenn es um Leib und Leben geht.

  Therac-25 (Linearbeschleuniger zur Anwendung in der Strahlentherapie): Ein schwerer Funktionsfehler kostete von Juni 1985 bis 1987 drei Patienten das Leben, drei weitere wurden schwer verletzt.

- Wo Sicherheit betroffen ist.

  Apple's SSL/TLS "goto fail bug": Eine duplizierte Codezeile führte dazu, dass die Überprüfung der Public-Key-Signatur übersprungen wurde und nie fehlschlug.

- Wenn großer finanzieller Schaden droht.

  Der **Pentium FDIV Bug** zwang Intel 1994 zu einer umfassenden Rückrufaktion; Kosten: 475 Mio $.

- ...

Was können wir tun?

- Fehler vermeiden
  durch Verwendung angemessener Programmiertechniken und -sprachen (Clean-Code, Entwurfsprinzipien …)
- Fehler finden
  Simulation und Testen
- Abwesenheit von Fehlern beweisen
  Statisch Code-Analyse (*linting*)
  Deduktive Methoden (Hoare-Logik)
- Fehler finden und Korrektheit beweisen
  Modellprüfverfahren (*model checking*)
  Maschinengestütztes Beweisen (*theorem proving*)

Simulation und Testen

Aufspüren von Fehlern in der Entwurfsphase (Simulation) oder im fertigen Produkt (Testen)

Methoden: Blackbox/Whitebox Testing, Coverage-Metriken etc.

👍 Erlaubt (offensichtliche) Fehler schnell und kosteneffektiv aufzudecken

👎 Unvollständig

- Keine Coverage-Metrik kann Fehlerfreiheit garantieren; eine Abschätzung, wie viele Fehler verbleiben, ist ebensowenig möglich
- Vollständige Abdeckung mit zunehmender Komplexität schwierig
- Nebenläufigkeit erschwert Tests

Statische Code-Analyse

Analysiert eine Approximation des Programms – *abstract interpretation*: symbolische Ausführung auf einer abstrakten Domäne (bspw. Intervalle)

Methoden: AbsInt, Astrée, …

👍

- Kann die Abwesenheit von Standardfehlern nachweisen
  Division durch 0, Out-of-bounds-Zugriffe, NULL-Pointer-Zugriffe, totes Coding, …
- Läuft vollautomatisch, oft sehr effizient auch auf großen Systemen

👎

- Kann (viele) Fehlmeldungen produzieren
- Analysiert nur Standardfehler

Deduktive Programm-Analyse

Programm-Analyse und Verifikation mithilfe formaler Semantik

Beispiel: Hoare-Logik (Schleifeninvarianten u.ä.)

👍
- Vollständig
- Aussagekraft nur durch den (menschlichen) Beweiser limitiert

👎
- Fehlerhafte Beweise bei manueller Durchführung möglich
  hier helfen Theorembeweiser
- Sehr schwer für nebenläufige/verteilte Systeme

Model Checking

Prüft den möglichen Zustandsraum eines formalen Modells gegen eine Spezifikation; wird kein Gegenbeispiel gefunden, erfüllt das Modell die Spezifikation

Gängige Logiken: LTL, CTL/CTL*, TLA+

👍
- Verifikation läuft automatisch
- Für reaktive, parallele und verteilte Systeme geeignet
- Erlaubt temporal-logische Aussagen (nicht nur *reachability*)
  never, *always*, *eventually*, *until*

👎
- Auf entscheidbare Probleme beschränkt: i.d.R. endliche Automaten
- Modellierung bedeutet Aufwand und produziert (mglw. hohe) Kosten
- Aussagen über ein abstraktes Modell, nicht über das System selbst

Maschinengestütztes Beweisen

Funktionale Programmierung als Modellierungssprache; interaktive (menschengeleitete) Beweiskonstruktion mit maschineller Verifikation

Beweisassistenten: Agda, Coq, ACL2, PVS, Isabelle …

👍

- Ausdrucksmächtig (Funktionale Programmierung)
  Erlaubt (theoretisch) die Formalisierung und Prüfung beliebiger Eigenschaften von beliebigen Programmen.
- Erlaubt das Erzeugen von *proof certificates*
- Teilweise umfangreiches weiteres Tooling
  Isabelle: *quickcheck* (Gegenbeispiel-Generator/Testfall-Generator), *sledgehammer* (Anbindung von *automated theorem provers*), Code-Generierung

👎

- Hoher manueller Aufwand (Modellierung und Beweiskonstruktion)
- Aussagen über ein abstraktes Modell, nicht über das System selbst

Zusammenfassung

Simulation und Testen können Fehler aufdecken, aber ihre Abwesenheit nicht belegen.

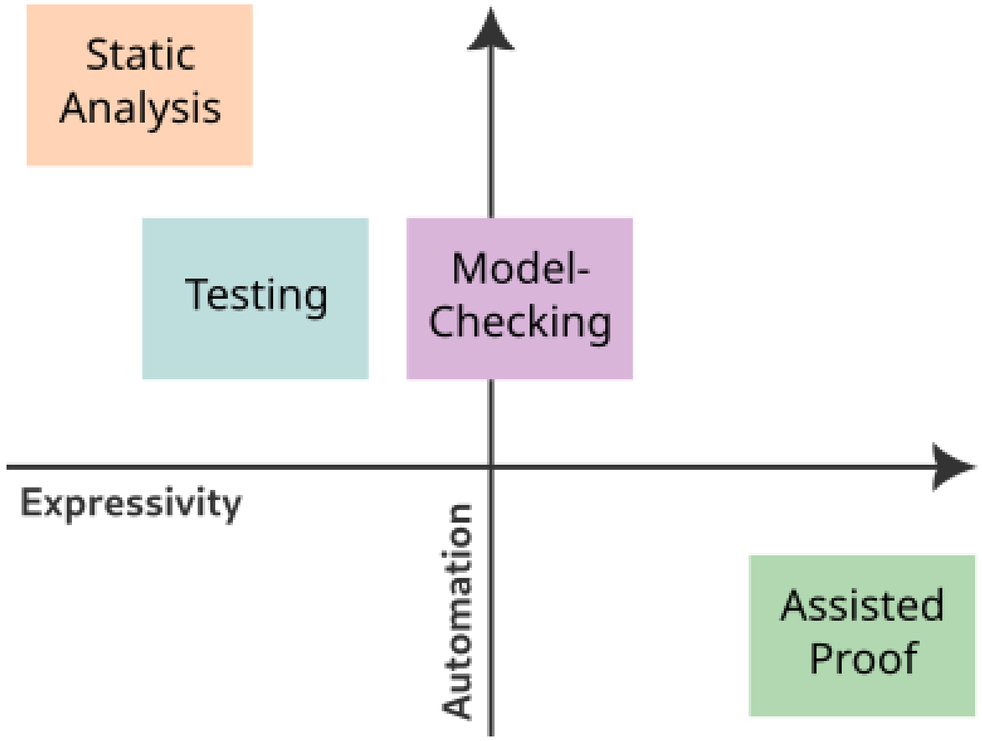
Sie betrachten eine *Teilmenge* aller möglichen Ausführungsszenarien.

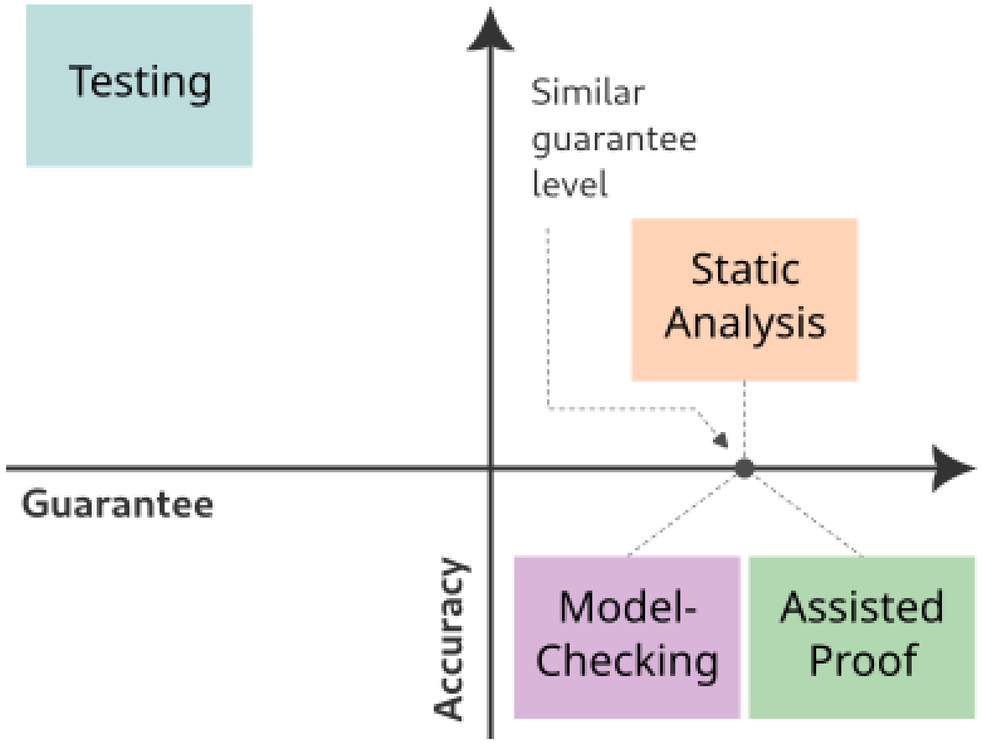
Deduktive Methoden und Code-Analyse können die Fehlerfreiheit nachweisen, produzieren aber u.U. Fehlmeldungen.

Sie betrachten eine *Obermenge* aller möglichen Ausführungsszenarien.

Model Checking und Beweisassistenten berücksichtigen alle (und nur die möglichen) Ausführungsszenarien, beziehen sich allerdings auf ein abstraktes Modell des Systems.

Zusammenfassung – Kosten und Nutzen





Wie interessant/aussagekräftig sind die nachgewiesenen Eigenschaften mit Blick auf (funktionale) Korrektheit?

Was kostet es?

Welche Garantien werden in Bezug auf die Korrektheit des *betrachteten* Systems gegeben?

Fazit

Formale Methoden sind kein Allheilmittel:

- Formal verifizierte Entwürfe müssen nicht funktionieren
  Fehler in der Spezifikation sind möglich.
  Das Modell kann unangemessen sein: relevante Aspekte werden "wegabstrahiert".
- Formale Verifikation kann zeitaufwendig und kostspielig sein

---

Formale Methoden sollten als Ergänzung zu Tests eingesetzt werden, nicht als ihr Ersatz!

Real-world Theorem Proving

- OpenJDK's TimSort-Algorithmus (KeY)

  de Gouw, S., de Boer, F.S., Bubel, R. et al.: Verifying OpenJDK's Sort Method for Generic Collections. In: Journal of Automated Reasoning 62, pp. 93–126 (2019).

- CompCert (Coq)

  Formal verifizierter Compiler für ein Subset von C99.

- seL4 (Isabelle)

  Formal verifizierter OS Microkernel.

Case-Study: seL4

Microkernel und TCB

OS-Kernel und Sicherheit/Stabilität

- Der Betriebssystemkernel läuft im privilegierten (Hardware-)Modus
- Kein Schutz bei Ausfällen/Fehlverhalten
  Jeder Bug kann potentiell beliebigen Schaden anrichten
- Gehört zwangsläufig zur *Trusted Computing Base* (TCB) eines Systems
  TCP = der Teil des Systems, der Security-Maßnahmen umgehen kann

Microkernel-Ansatz

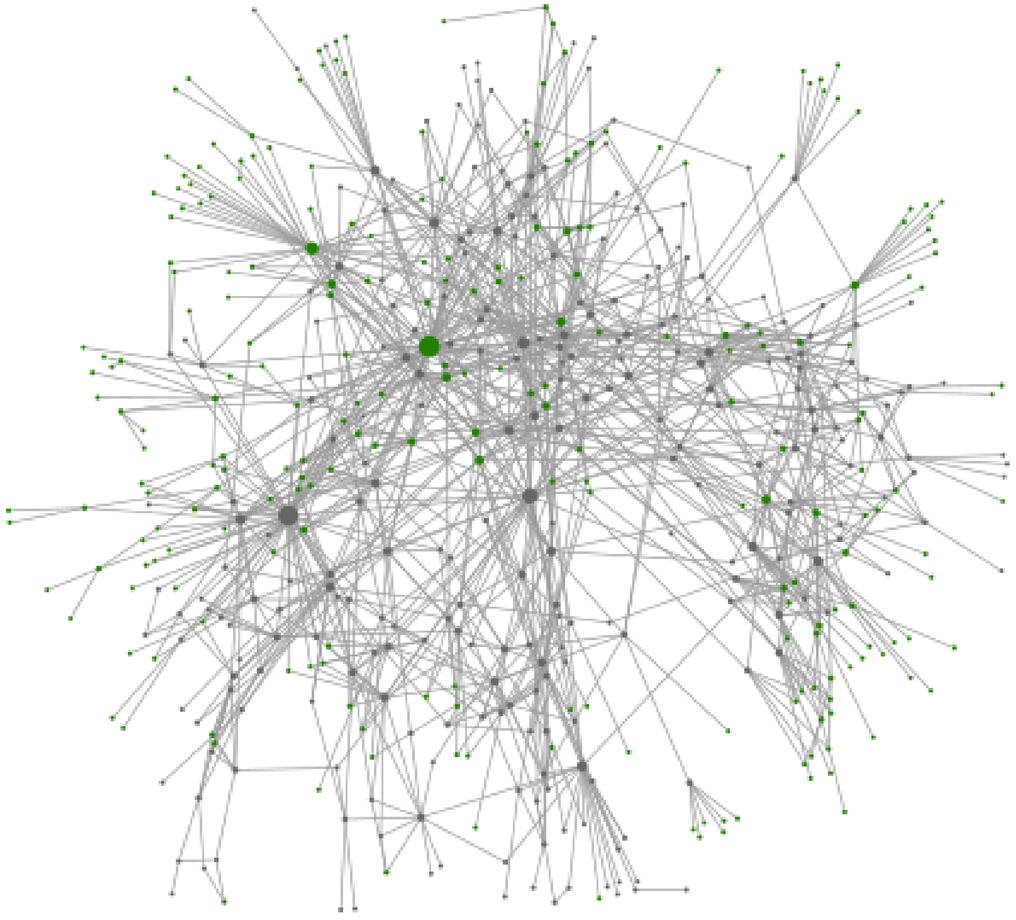
- Gegenüber (traditionellen) monolithischen Designs stark reduzierter Funktionsumfang: Kernel enthält nur das absolute Minimum
  Minimaler Wrapper um Hardware-Funktionalitäten, die eine privilegierte Ausführung erfordern
  Alle weiteren OS-Services sind als normale Programme implementiert und laufen im (unprivilegierten) User-Modus.

Vertrauenswürdigkeit vs. Komplexität

Das Problem der Vertrauenswürdigkeit des Kernels wird aufgrund des geringeren Funktionsumfangs handhabbarer.

Um ausreichende/konkurrenzfähige Performance sicherzustellen, nimmt die Komplexität solcher Systeme aber zu.

Aufrufgraph des seL4 Kernels. Knoten stellen Funktionen und Kanten Funktionsaufrufe dar.



Quelle:
Gerwin Klein et al.
seL4: Formal Verification of an Operating-System Kernel.
https://doi.org/10.1145/1629575.1629596

Machbarkeit mit vertretbarem Aufwand

Der relativ kleine Funktionsumfang macht Verifikation überhaupt erst (mit vertretbarem Aufwand) möglich.

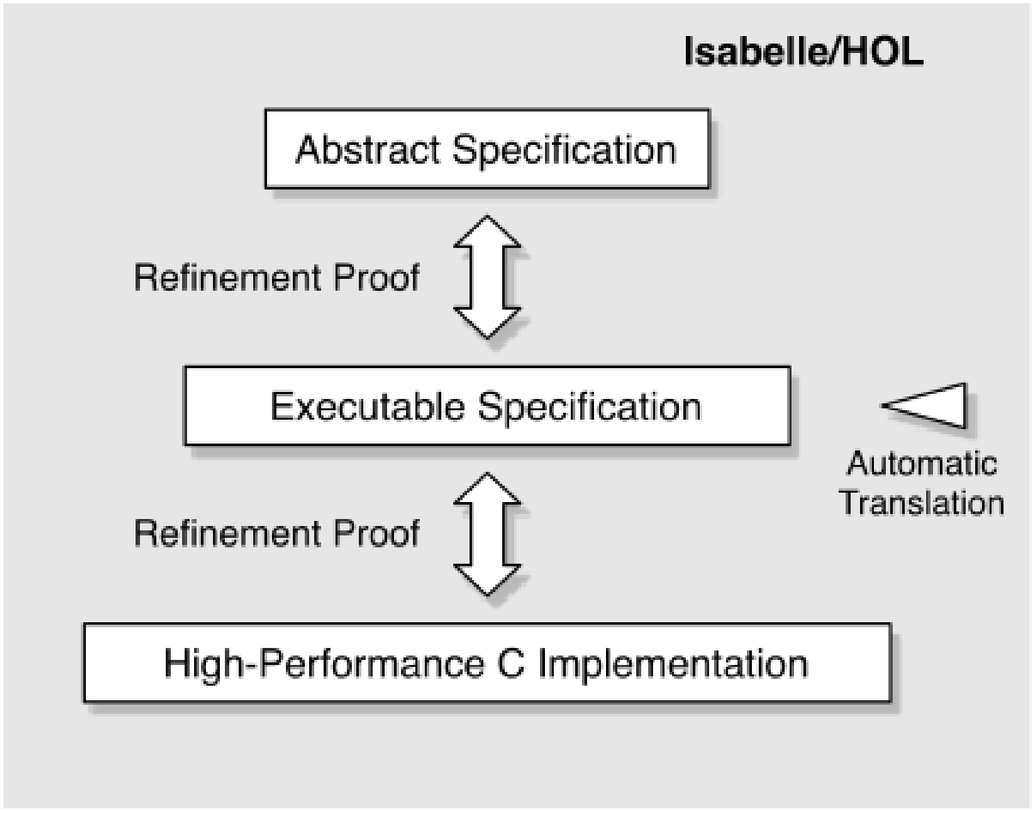
Die Verifikation erlaubt sehr starke Aussagen, insb. in Hinblick auf die Vertrauenswürdigkeit des Systems:

no code injection attacks, no buffer overflows, no null pointer access, no ill-typed pointer access, no memory leaks, no non-termination, no arithmetic or other exceptions, no unchecked user arguments, aligned objects, wellformed data structures, algorithmic invariantes, correct book-keeping

Kernel Design for Verification

- Kernel-Entwicklung from scratch
- Kernel/Proof Co-Design
  Iteratives Kernel-Design mit Haskell als *intermediate target*.
  OS-Entwickler konnten die Funktionalitäten des Systems ohne Einschränkungen entwickeln. Gleichzeitig stand ein verifikationsfreundliches Artefakt zur Verfügung, das in den Beweisassistenten importiert werden konnte und dort als *intermediate executable specification* diente.
- Verifikationsfreundliche Entwurfsentscheidungen
  Umgang mit globalen Variablen und Seiteneffekten (nur explizit; vereinfacht durch die Verwendung von Haskell); Kernel-Speicher-Management in authorisierte Anwendung ausgelagert (primär entwicklungstechnisch sinnvoll – Buchhaltung – aber auch für die Verifikation günstig); Vermeidung von Nebenläufigkeit und Nicht-Determinismus und Beschränkung der Verifikation auf die Single-Prozessor-Variante von seL4; Übersetzung von I/O-Interrupts von Treibern in Messages zur Reduktion von Komplexität

Drei Refinement-Layers



1. Abstract Specification:
   Was tut das System?
2. Executable Specification
   Wie arbeitet das System?
   (ohne Optimierungsaspekte)
3. High-Performance C Implementation
   Manuell erstellter C-Code
   präzise Modellierung der C-Semantik *from first principles*

Quelle:
Gerwin Klein et al.
seL4: Formal Verification of an Operating-System Kernel.
https://doi.org/10.1145/1629575.1629596

Zu guter Letzt ein paar Zahlen

	Haskell/C	**Isabelle**	**Invariants**	**Proof**
	pm / kloc	**kloc**		**py / klop**
abst.	4 / --	4.9	~ 75	8 / 110
exec.	24 / 5.7	13	~ 80	3 / 55
impl.	2 / 8.7	15	0	

- Die Formalisierung umfasst insgesamt 200 000 Zeilen Isabelle-Skript-Code.
- Ca. 30 Personenmonate (pm) wurden für die abstrakte Spezifikation, den Haskell Prototypen und die C-Implementierung aufgewendet.
  Inklusive Design, Dokumentation, Entwicklung und Testing.
- Zu den oben aufgeführten 11 Personenjahren (py) für die Beweisentwicklung kamen noch 9 py an Forschung und Vorbereitung
  Formale Sprachframeworks, Beweiswerkzeuge, Beweisautomatisierung, Theorembeweisererweiterungen und -bibliotheken
- Schätzung für eine Wiederholung nach dem gleichen Schema: 6 py bzw. 8 py für Kernelentwicklung und Beweise

Verifikation mit Isabelle/HOL

Joachim Breitner: Verifikation mit Isabelle – BobKonf 2016

Was ist Isabelle?

Isabelle ist...

- ein interaktiver Theorembeweiser
- eine funktionale Programmiersprache
  mit algebraischen Datentypen, rekursiven Funktionen, Typklassen...
  aus der man Code in Haskell, Scala, Standard ML, OCaml generieren kann.
- eine Entwicklungsumgebung für Programme *und* Beweise
  basierend auf jEdit.

Programmverifikation mit Isabelle – mehrere Ansätze

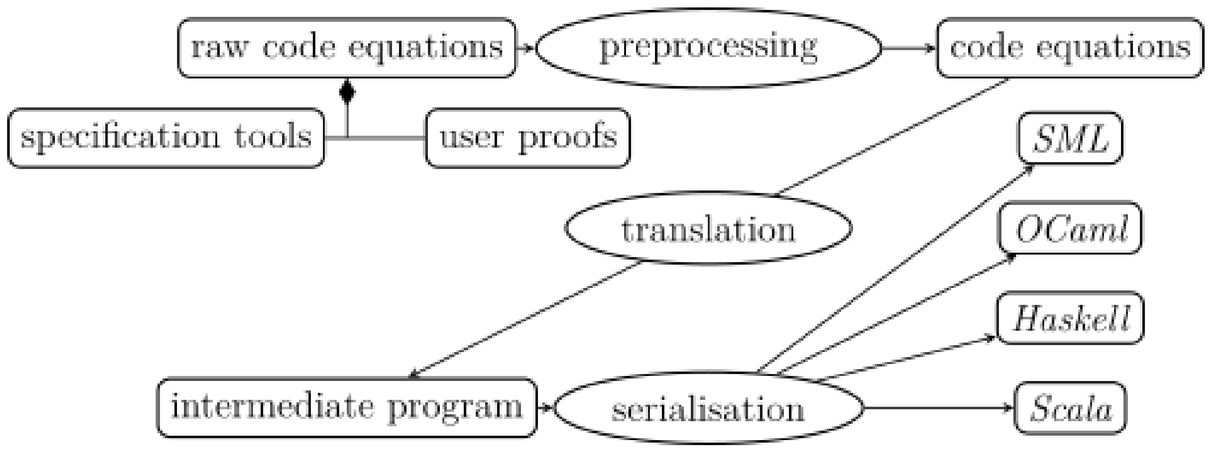
Haskell (o.ä.) → Isabelle

… als *shallow embedding*:
Haskell-Funktionen werden Isabelle-Funktionen

… als *deep embedding*:
Haskell-Syntax wird über einen Isabelle-Datentyp dargestellt.

Isabelle → Haskell (o.ä.)

per Code-Generator



Quelle:
Florian Haftmann, Lukas Bulwahn.
Code generation from Isabelle/HOL theories.

Unser Plan

1. Einen einfachen Datentypen definieren (Listen)
2. Einfache Funktionen darüber definieren:

   `app` und `reverse`
3. Daraus Haskell-Code generieren
4. Etwas über unseren Code beweisen:

   ```
   reverse (reverse xs) = xs
   ```
5. Eine komplexere Datenstruktur implementieren (Queues)
6. Mit *program refinement* die Implementierung verbessern
7. Falls noch Zeit ist: Die Datenstruktur verifizieren

Die ersten Schritte

Als Erstes legen wir eine neue Datei `Queue.thy` an.

Grundgerüst:

```
theory Queue
imports Main
begin

end
```

Funktionales Programmieren in Isabelle/HOL

Wir setzen folgendes Haskell-Programm in Isabelle um:

```haskell
data List a = N | C a (List a)

app :: List a -> List a -> List a
app N ys = ys
app (C x xs) ys = C x (app xs ys)

reverse :: List a -> List a
reverse N = N
reverse (C x xs) = app (reverse xs) (C x N)
```

Code-Generierung in Isabelle

Um Haskell-Code zu generieren, schreiben wir

```
export_code reverse in Haskell file "out/"
```

ans Ende der Datei.

Die Listen-Definition und app werden als Abhängigkeiten automatisch exportiert.

Im Verzeichnis `out/` sollte nun eine Datei `Queue.hs` liegen, die man bspw. mit `ghci` laden kann.

Hinweis: Wir verwenden hier ein Legacy-Feature (`... file "path"`). Ohne den `file`-Zusatz wird der Code erst einmal nur ins logische File-System innerhalb des Theorie-Kontextes geschrieben. Nach dem Kompilieren der Theorie kann er extrahiert werden.

Unser erster Beweis

Beweismethode: Induktion + Simplifikation

Struktur (in unseren Fällen):

```
lemma name[simp]: "ausdruck1 = ausdruck2"
  apply (induction xs)
  apply auto
  done
```

Was wollen wir zeigen?

```
lemma reverse_reverse[simp]: "reverse (reverse xs) = xs"
```

Hierfür werden wir weitere Hilfslemmas benötigen. Welche?

Amortized Queue

Wir ergänzen den folgenden Code:

```haskell
data AQueue a = AQueue (List a) (List a)

emptyQ :: AQueue a
emptyQ = AQueue N N

enqueue :: a -> AQueue a -> AQueue a
enqueue x (AQueue xs ys) = AQueue (C x xs) ys

dequeue :: AQueue a -> (Maybe a, AQueue a)
dequeue (AQueue N N) = (Nothing, AQueue N N)
dequeue (AQueue xs (C y ys)) = (Just x, AQueue xs ys)
dequeue (AQueue xs N) =
  case reverse xs of C y ys -> (Just y, AQueue N ys)
```

Der Code ist zu lahm

Unsere Definition von `reverse` ist zwar verifikations-freundlich, aber nicht performant ($O(n^2)$).

Besser ist:

```
fast_rev :: List a -> List a -> List a
fast_rev N ys = ys
fast_rev (C x xs) ys = fast_rev xs (C x ys)
```

Dies könnten wir wie folgt nutzen:

```
reverse :: List a -> List a
reverse xs = fast_rev xs N
```

Schnellerer Code

Aber wir wollen unsere Beweise zu `reverse` nicht ändern, sondern nur die Implementierung von `reverse` in dequeue!

Wir beweisen zuerst, wie sich `fast_rev` verhält:

```
lemma fast_rev_reverse[simp]: "fast_rev xs ys = app (reverse xs) ys"
  <proof>
```

Achtung: Induktionsbeweis geht nicht ohne Weiteres durch.

Dann lassen wir den Code-Generator `reverse` durch `fast_rev` ersetzen:

```
lemma [code_unfold]: "reverse xs = fast_rev xs N"
  <proof>
```

Wie beeinflusst das den generierten Code?

Queue verifizieren

Um die Queue selbst zu verifizieren suchen wir nach einem äquivalenten, aber einfacheren Datentypen (Performance ist egal).

Listen!

Wir implementieren `emptyQ'`, `enqueue'` und `dequeue'` auf `'a List`, so dass sie "offensichtlich korrekt" sind.

Und beschreiben eine Queue als Liste:

```
fun list_of :: "'a AQueue -> 'a List" where ...
```

Schließlich zeigen wir: Die alten Methoden machen "dasselbe" wie die neuen.
Die Beweise über die Queue werden ggf. komplizierter:

```
apply (induction q rule: dequeue.induct)
apply (auto split: List.split)
done
```

Tipp: Wir brauchen hier ein Hilfslemma der Form `(app xs ys = N) = (...)`.

Wie geht es weiter?

- Isabelle kennenlernen, z.B. mit
  Nipkow, Klein: *Concrete Semantics*
- Den Code-Generator kennenlernen, z.B. mit
  Florian Haftmann, Lukas Bulwahn: *Code generation from Isabelle/HOL theories*
- Mehr auf:
  https://isabelle.in.tum.de/documentation.html
- Das Archive of Formal Proofs durchstöbern
  https://www.isa-afp.org
- Auf das iSAQB-Modul "Formale Methoden" warten
  Das Curriculum entwickeln wir bei der Active Group gerade. Neuigkeiten dazu werden wir in unserem Blog ankündigen (siehe nächste Folie).

Active Group GmbH

- Projektentwicklung mit funktionaler Programmierung
- Scala, Clojure, F#, Haskell, OCaml, Erlang, Elixir, Swift
- **Schulungen und Coaching**

Blog: https://funktionale-programmierung.de

Referenzen und Dank

Danke für die Aufmerksamkeit 😊

Happy proving!

Den Workshop-Inhalt verdanke ich Joachim Breitner 🙏

Joachim Breitner: *Verifikation mit Isabelle* – BobKonf 2016
https://www.joachim-breitner.de/publications/Isabelle_BobKonf2016_2016-02-19-slides.pdf

Die Code-Beispiele sind inspiriert von:

Florian Haftmann, Lukas Bulwahn: *Code generation from Isabelle/HOL theories*.
Tobias Nipkow: *Programming and Proving in Isabelle/HOL*.