Interactive Formal Verification

Tjark Weber

Docentship Lecture
2024-02-14

Overview

- Motivation
- Interactive theorem proving
- Proof assistants
- Reasoning about an imperative language
- Conclusion

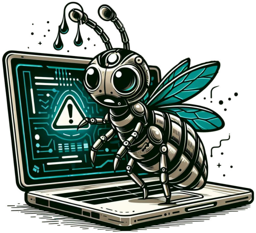
Motivation

Complexity Breeds Bugs

Modern hardware and software systems can be extremely complex.

- Apple's M2 Ultra: 134 billion transistors
- Linux kernel: $>$ 27 million lines of code

Complex systems almost inevitably contain bugs.



Rocket Science

On its first test flight in June 1996, the European Ariane 5 space rocket (worth nearly US$ 400 million) exploded 37 seconds after launch because of a malfunction in its control software.

The software was originally written for the Ariane 4 and could not cope with the higher speed of the Ariane 5 rocket.

Reliability Matters

The annual cost of poor software quality in the US has been estimated at 2.4 trillion dollars.[1]

Software controls nuclear power plants, defense systems, aircraft, railway signals and many other safety-critical systems.

Some systems are explicitly designed to prevent human intervention.

[1] Consortium for Information & Software Quality (CISQ): *The Cost of Poor Software Quality in the US: A 2022 Report*

Testing?

Can't we just test these systems until we are sure that they work correctly?

Testing?

Can't we just test these systems until we are sure that they work correctly?

- Exhaustive testing is infeasible for complex systems.
- Critical systems (avionics, . . .) are required to meet a standard of 10^{-9} failures per hour. Testing to such a standard is infeasible.

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsger W. Dijkstra

Interactive Theorem Proving

A Solution: Formal Verification

Prove (or disprove) **the correctness** of hardware and software systems based on

- a formal semantics,
- with respect to a formal specification,
- using formal (mathematical) methods.

What is Interactive Theorem Proving?

- Working in a logical formalism ...
  - with precise definitions of concepts
  - and a formal deductive system
- ... supported by a proof assistant ...
  - that checks the correctness of each step
- ... to construct hierarchies of definitions and proofs
  - libraries of formalized mathematics
  - specifications of components and properties

Example: Reasoning About Finite Sequences

```
datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

fun reverse :: "'a seq ⇒ 'a seq"
where
  "reverse Empty = Empty"
| "reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)"

lemma conc_empty: "conc xs Empty = xs"
  by (induct xs) simp_all

lemma conc_assoc: "conc (conc xs ys) zs = conc xs (conc ys zs)"
  by (induct xs) simp_all

lemma reverse_conc: "reverse (conc xs ys) = conc (reverse ys) (reverse xs)"
  by (induct xs) (simp_all add: conc_empty conc_assoc)

lemma reverse_reverse: "reverse (reverse xs) = xs"
  by (induct xs) (simp_all add: reverse_conc)
```

Source: Isabelle2023

What is Interactive Theorem Proving Like?

- Can prove hard theorems
- Time consuming, occasionally frustrating, but also rewarding
- Potentially addictive
- Erodes trust in informal proofs

Some Landmark Projects

- L4.verified (G. Klein et al., 2009): functional correctness of the seL4 microkernel (about 8700 lines of C) — using Isabelle
- CompCert (X. Leroy et al., 2009): a formally verified compiler for (almost all of) the C language — using Coq
- Flyspeck (T. Hales et al., 2014): a formal proof of the Kepler conjecture — using Isabelle and HOL Light

Proof Assistants

Proof Assistants

Proof assistants are software tools that assist with the development of formal (machine-readable) specifications and proofs.

ACL2 Agda Coq HOL4

Isabelle Lean Mizar PVS

. . . and many others

Proof Assistants by Logical Formalism

- Based on higher-order logic
  - **Isabelle**, HOL (many versions), PVS
- Based on constructive type theory
  - **Coq**, Twelf, Agda, Lean
- Based on other formalisms
  - ACL2 (first-order logic with recursion), Mizar (set theory)

Higher-Order Logic

- First-order logic extended with quantification over functions and predicates
- No distinction between terms and formulas
- Polymorphic types (e.g., α seq)
- Functional programming: datatypes, recursive functions, ...

"HOL = Functional programming + Logic" [2]

[2] Tobias Nipkow: *Programming and Proving in Isabelle/HOL*

Key Features of Proof Assistants

- Logical formalism (higher-order logic, type theory, etc.)
- Operation and control
  - User interface / proof language
  - Automation
- Libraries of formalized mathematics
- Tools: library search, typesetting, code generation, . . .

Reasoning About an Imperative Language

Source: Isabelle2023

Commands

Concrete syntax:

$$
\begin{array}{rcl}
\mathit{com} & ::= & \texttt{SKIP} \\
 & | & \mathit{vname} ::= \mathit{aexp} \\
 & | & \mathit{com} \;;;\; \mathit{com} \\
 & | & \texttt{IF}\ \mathit{bexp}\ \texttt{THEN}\ \mathit{com}\ \texttt{ELSE}\ \mathit{com} \\
 & | & \texttt{WHILE}\ \mathit{bexp}\ \texttt{DO}\ \mathit{com}
\end{array}
$$

Commands

Abstract syntax:

$$\begin{array}{lcl} \textbf{datatype}\ \mathit{com} & = & \mathrm{SKIP} \\ & | & \mathrm{Assign}\ \mathit{vname}\ \mathit{aexp} \\ & | & \mathrm{Seq}\ \mathit{com}\ \mathit{com} \\ & | & \mathrm{If}\ \mathit{bexp}\ \mathit{com}\ \mathit{com} \\ & | & \mathrm{While}\ \mathit{bexp}\ \mathit{com} \end{array}$$

Com.thy

Com.thy

Big-step Semantics

Concrete syntax:

$$(\mathit{com}, \mathit{initial\text{-}state}) \Rightarrow \mathit{final\text{-}state}$$

Intended meaning of $(c, s) \Rightarrow t$:

command c started in state s terminates in state t

Big-step Semantics

Logically, the notation

$$(c, s) \Rightarrow t$$

is just infix syntax for

$$\textit{big\_step}\ (c, s)\ t$$

where

$$\textit{big\_step} :: \textit{com} \times \textit{state} \Rightarrow \textit{state} \Rightarrow \textit{bool}$$

is an inductively defined predicate.

Big_Step.thy

Big_Step.thy

Reasoning About Programs and Languages

Having defined syntax and semantics of a programming language, we can prove theorems about

- the behaviour of individual programs,
- properties of the semantics (e.g., determinism).

Hoare_Examples.thy

Hoare_Examples.thy

Conclusion

Benefits of Interactive Formal Verification

- Rigorous correctness proofs: **the system works!**
- Certification: industry standards (e.g., Common Criteria) require formal methods for higher assurance levels
- Knowledge representation: formal specifications serve as precise documentation of system requirements
- Long-term maintenance: incremental system changes typically require only incremental changes to proofs

Current Research Challenges

- Automation
  - Proof exchange
  - Auto-formalization
- Formalization
  - Undergraduate material
  - Recent research results
  - Verified proof assistants
- User interfaces
  - Management of large formal libraries
  - Proof languages

Strengths and Weaknesses

Interactive theorem proving is **more laborious** than other formal verification techniques . . .

. . . but it allows to reason about (almost) anything with extremely high levels of assurance, and to prove **hard theorems** that are currently far beyond the reach of any other technique.

> *"Beware of bugs in the above code; I have only proved it correct, not tried it."*
>
> *Donald Knuth*

Questions?

The LCF Architecture

- A small **kernel** implements the logic. Only kernel functions can generate new theorems.
- All specification methods and automatic proof procedures generate full proofs by invoking kernel functions.
- Unsoundness is less likely with this architecture ...
- ... but the implementation (of specification methods and automatic proof procedures) is more complicated.