Degrees of Software Correctness

Uniting the Spectrum of Verification

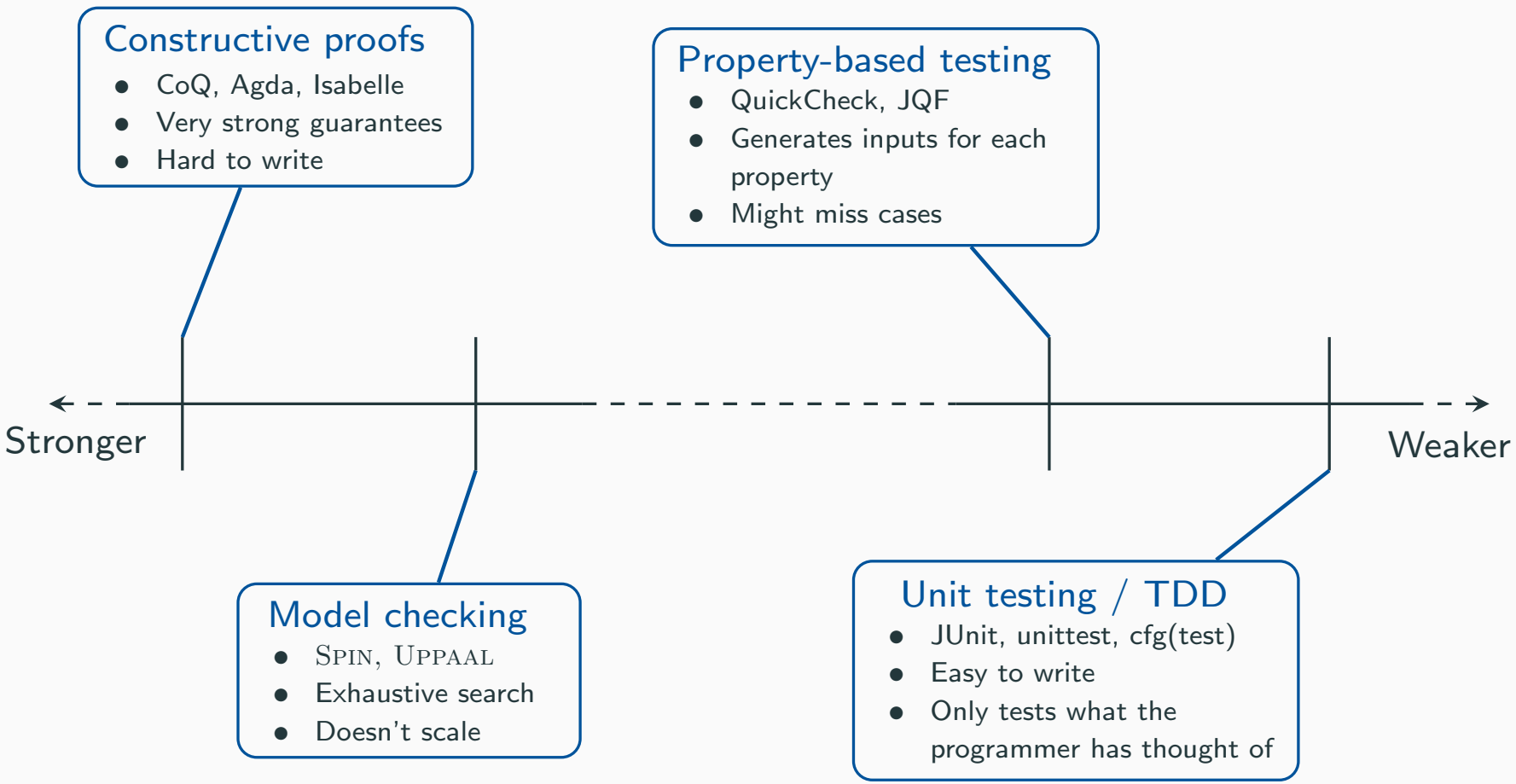
Thomas Ekström Hansen

10th November 2023

Introduction

- We generally want software to be correct
- Many approaches to this
  - Test Driven Development (TDD)
  - Mutation testing
  - Property-based testing
  - Model Checking
  - Type theory
  - Mathematical abstractions and proofs
- No one-size-fits-all
  - So compromises are made

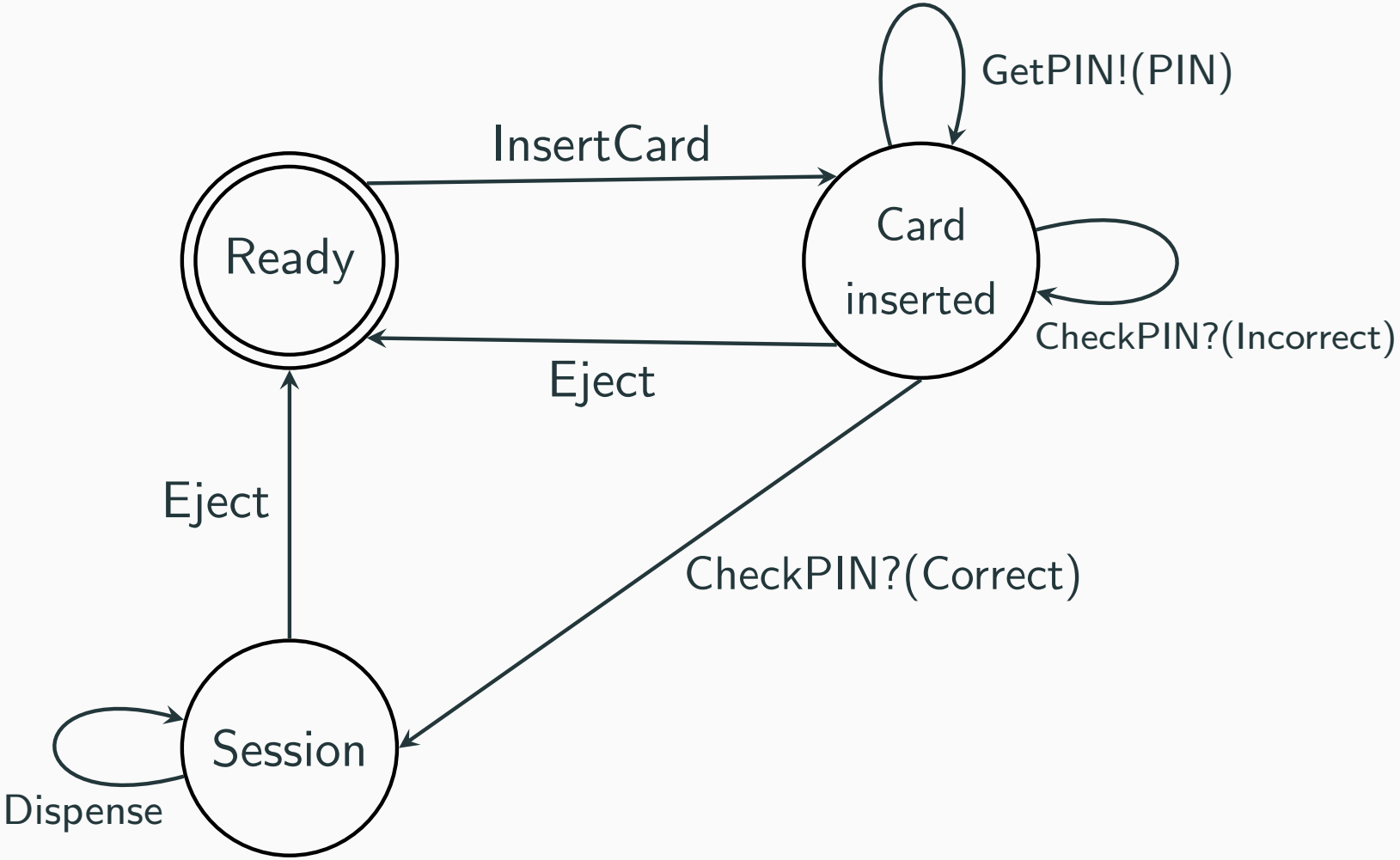
Spectrum of Verification



The eternal problem with verification systems

- All verification systems face the same problem: ergonomics
- If the system is not unobtrusive, people are unlikely to use it
- Our hope is that our approach tries to not get in the way by being part of the program
  - Compiler and type-system assist you rather than hinder
  - Verify as you go along, tuning the strictness as necessary
  - Escape hatches provided for prototyping

Visuals can really help



State machine of an ATM

Using types only gets you part of the way there

```
data State  =  Ready | CardInserted | Session

data CheckPINRes  =  Incorrect | Correct

data CMD : (ty : Type) -> State -> (ty -> State) -> Type
  where
  InsertCard : CMD () Ready (const CardInserted)
  Dispense   : CMD () Session (const Session)
  GetPIN     : CMD () CardInserted (const CardInserted)

  CheckPIN   : (Vect 4 Nat)
            -> CMD CheckPINRes CardInserted
                   (\res => case res of
                                Incorrect => CardInserted
                                Correct => Session)

  GetAmount  : CMD Nat state (const state)

  EjectCard  : CMD () state (const Ready)
```

Using types only gets you part of the way there

Accepted by the compiler:

```
atmProg : CMD ()
atmProg =
  do InsertCard
     pin <- GetPIN
     Correct <- CheckPIN pin
       | Incorrect => EjectCard
     amt <- GetAmount
     Dispense amt
     EjectCard
```

Rejected by the compiler:

```
badProg : CMD ()
badProg =
  do InsertCard
     pin <- GetPIN
     Correct <- CheckPIN pin
       | Incorrect => InsertCard
     amt <- GetAmount
     Dispense amt
     -- forgot to eject
```

Disadvantages of the `CMD`-like approach

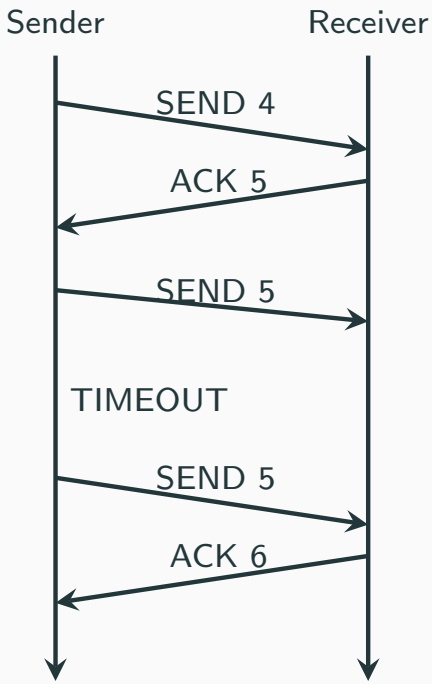
- Not the most ergonomic
  - It is a Domain Specific Language
  - So it needs an interpreter; it is not directly runnable code
  - The type declarations can get lengthy
- Composition is currently undefined
  - The theory it builds on does not currently support composition
  - But we often want to compose code and functionality

Benefits of a multifaceted approach

1. Adaptability — right tool for the job
2. Speed — can trade speed for level of verification
3. **Coherence** — all done in one system
    - No need to translate to model-checking tool
    - Specification lives alongside model lives alongside implementation
    - The implementation is just there; it *is* runnable code
    - Parts can be verified independently *while* combining into an overall system

Example: ARQ – Overview

- Automated Repeat reQuest (of course -_-)
- Sender sends one packet at a time
- Wait for the receiver to acknowledge they are ready for the next packet
- Resend if no acknowledgement is received within a given timeout



Sample ARQ transmission

Example: ARQ – Verification challenges

Different parts necessitate different amounts of verification

- Data verification is relatively simple
  - Proofs might be feasible.
- Potentially infinite packet numbers
  - Proofs will be tedious
  - Not fit for model checking
- Network interface possibly outwith our control
  - Likely using a library or similar
  - Cannot control how interfaces behave
- Other half likely not even physically local
  - Interfacing with someone else's code and/or infrastructure
  - Impossible to reason about beyond handling responses

Example: ARQ – Our approach

- Compression can be proven to be correctly implemented
  - Types for compressed and uncompressed data
  - Link the two via `` `compress` `` and `` `uncompress` `` functions
  - The compiler can then help us implement those correctly
- QuickCheck can simulate network behaviour
  - Generate an arbitrary packet
  - Simulate sending with a chance of failure
  - Check that 10 or 100 different sequences all get delivered
  - All done at compile-time, using the same datatypes as the implementation
- Some unit tests could be added for the network implementation
  - Interaction with external libraries, so unit tests is best we can do
  - Confidence from QuickCheck increased by these passing

Example: ARQ – Benefits of our approach

- All in the same system
- We do not need to trust we have modelled the problem correctly, we can test that!
- Can use as stronger or weaker verification as wanted/required
- There is a link between the verified and implemented model: the datatypes!
- The compiled program is an executable version of our verified implementation!

Conclusion

- Many verification tools exist, none of them cover enough on their own
- Instead of “competing”, we combine the systems to work together
- Hopefully this will lead to wider adoption and better whole-system soundness

Questions

Questions?