Verifying an Operating System Kernel

Michael Norrish

Australian Government
Department of Communications, Information Technology and the Arts
Australian Research Council

NICTA Members
ANU THE AUSTRALIAN NATIONAL UNIVERSITY
UNSW
NSW First for Business
Department of State and Regional Development
BusinessACT ACT GOVERNMENT

Victoria The Place To Be
THE UNIVERSITY OF MELBOURNE
The University of Sydney
Queensland Government
Griffith UNIVERSITY
QUT Queensland University of Technology
THE UNIVERSITY OF QUEENSLAND AUSTRALIA
NICTA Partners

```
                             Windows

An exception  06 has occured at 0028:C11B3ADC in VxD DiskTSD(03) +
00001660.  This was called from 0028:C11B40C8 in VxD voltrack(04) +
00000000.  It may be possible to continue normally.

*  Press any key to attempt to continue.
*  Press CTRL+ALT+RESET to restart your computer.  You will
   lose any unsaved information in all applications.

                      Press any key to continue
```

The Problem

seL4 + L4.verified

Goals:

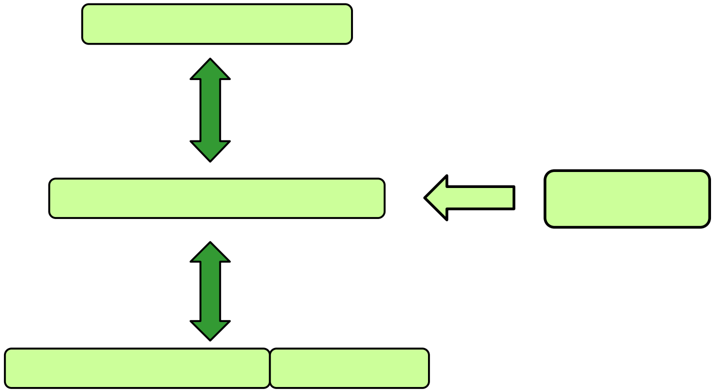
- Formal specification of kernel and machine
- Verified production quality, high-performance kernel

Address problems in L4:

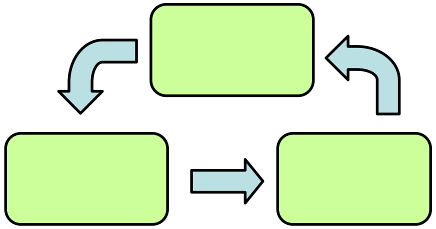
- Communication control
- Kernel resource accounting
- No performance penalty for new features
  - 30 cycles per syscall ok. Maybe.

Overview

- The seL4 Kernel
  - Interface
  - State
  - Kernel Objects



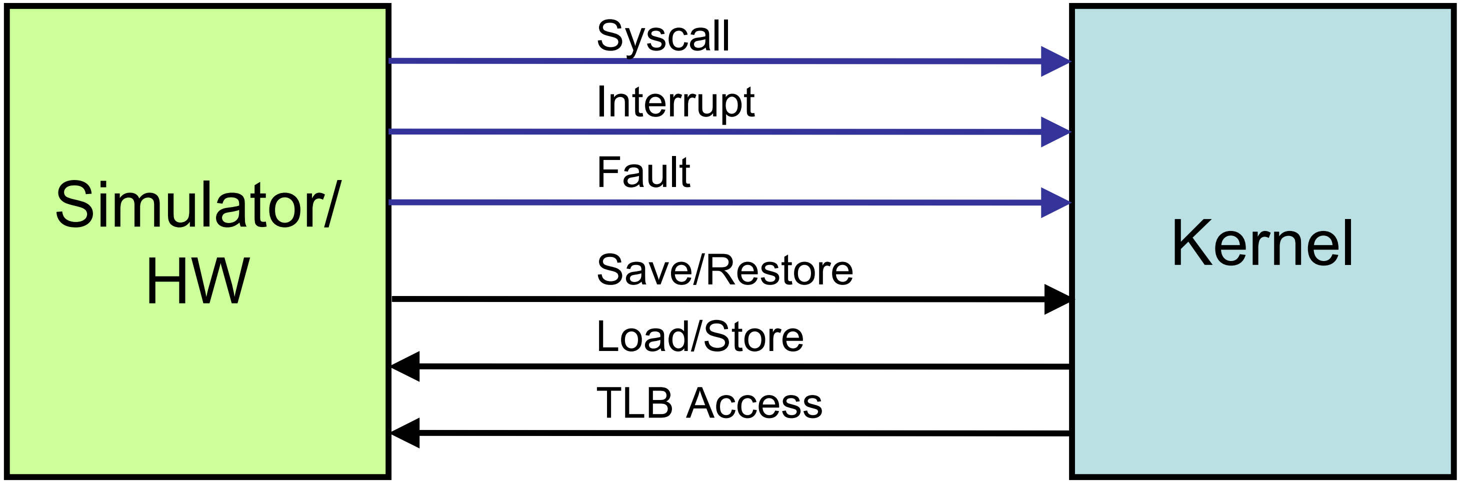
- Interesting Problems
  - Designing and formalizing an OS kernel
  - Refinement on monadic functional programs



seL4

secure embedded L4

Kernel Interface



Simulator/
HW

Syscall

Interrupt

Fault

Save/Restore

Load/Store

TLB Access

Kernel

- Kernel is a state transformer:

```
kernel :: Event ) KernelState ) KernelState
```

Kernel State

- Physical memory

  Storage: **`obj_ref ) kernel_object option`**

- Mapping database

  Capability derivations: **`cte_ref ) cte_ref option`**

- Current thread

  Pointer: **`obj_ref`**

- Machine context

  Registers, caches, etc

Kernel Objects (simplified)

- Capability Table

```
cap_ref ) capability
```

- Thread Control Block (TCB)

```
record    ctable, vtable :: capability
          state :: thread_state
          result_endpoint, fault_endpoint :: cap_ref
          ipc_buffer :: vpage_ref
          context :: user_context
```

- Endpoint:

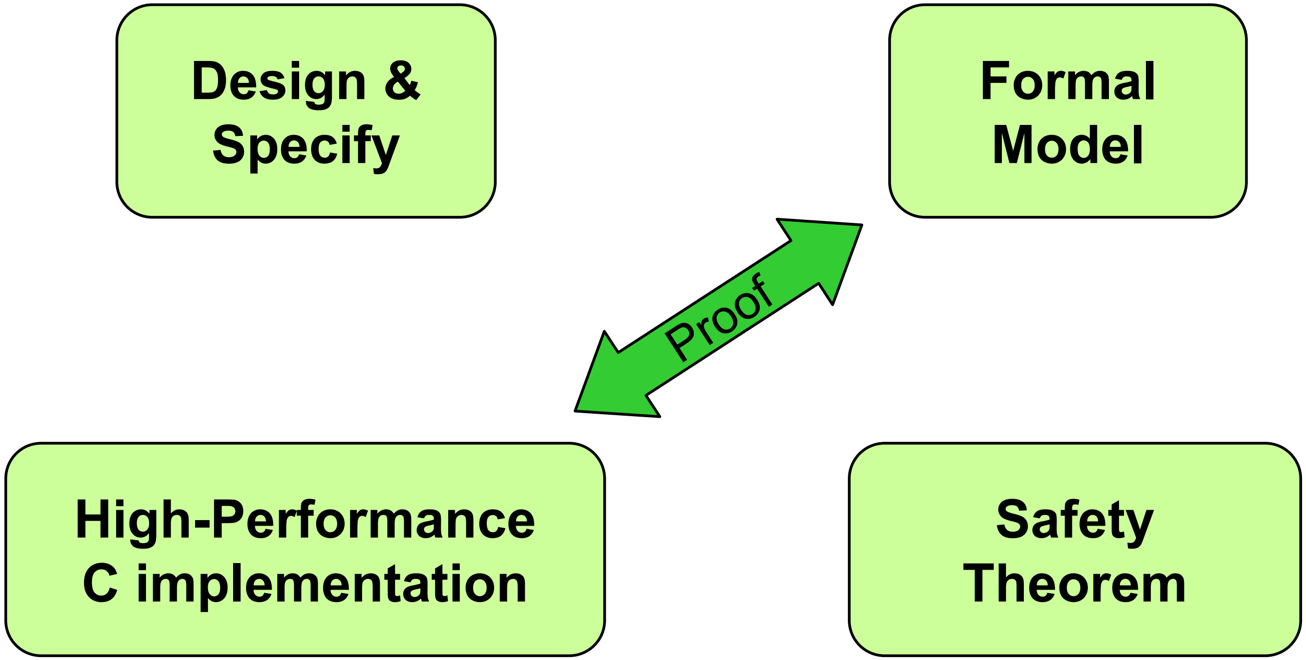
```
Idle | Receive (obj_ref list) | Send (obj_ref list)
```

- Data Page

Designing and Formalising

concrete syntax is everything

How to design and formalise a new kernel



Design & Specify

Formal Model

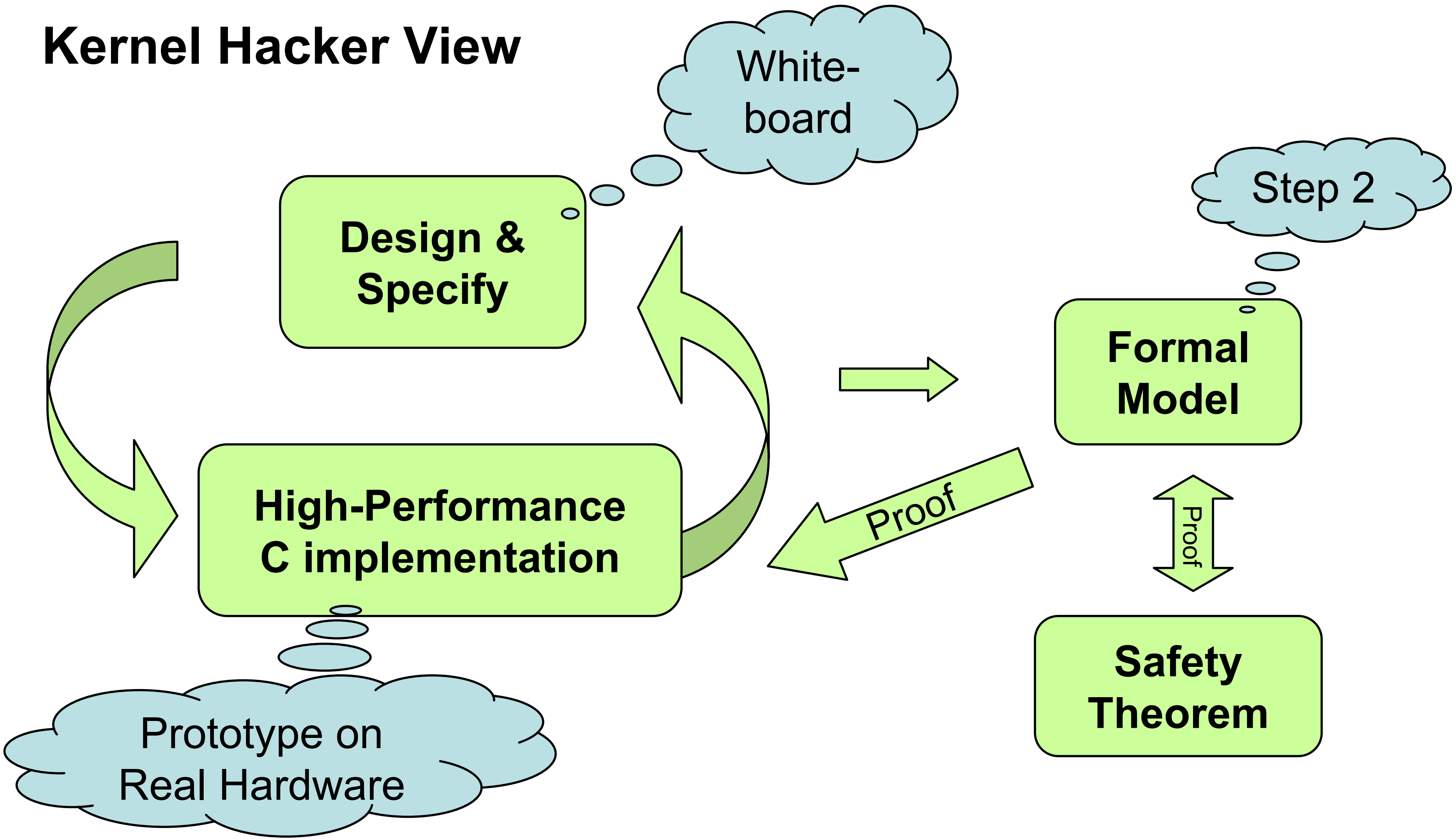
Proof

High-Performance C implementation

Safety Theorem

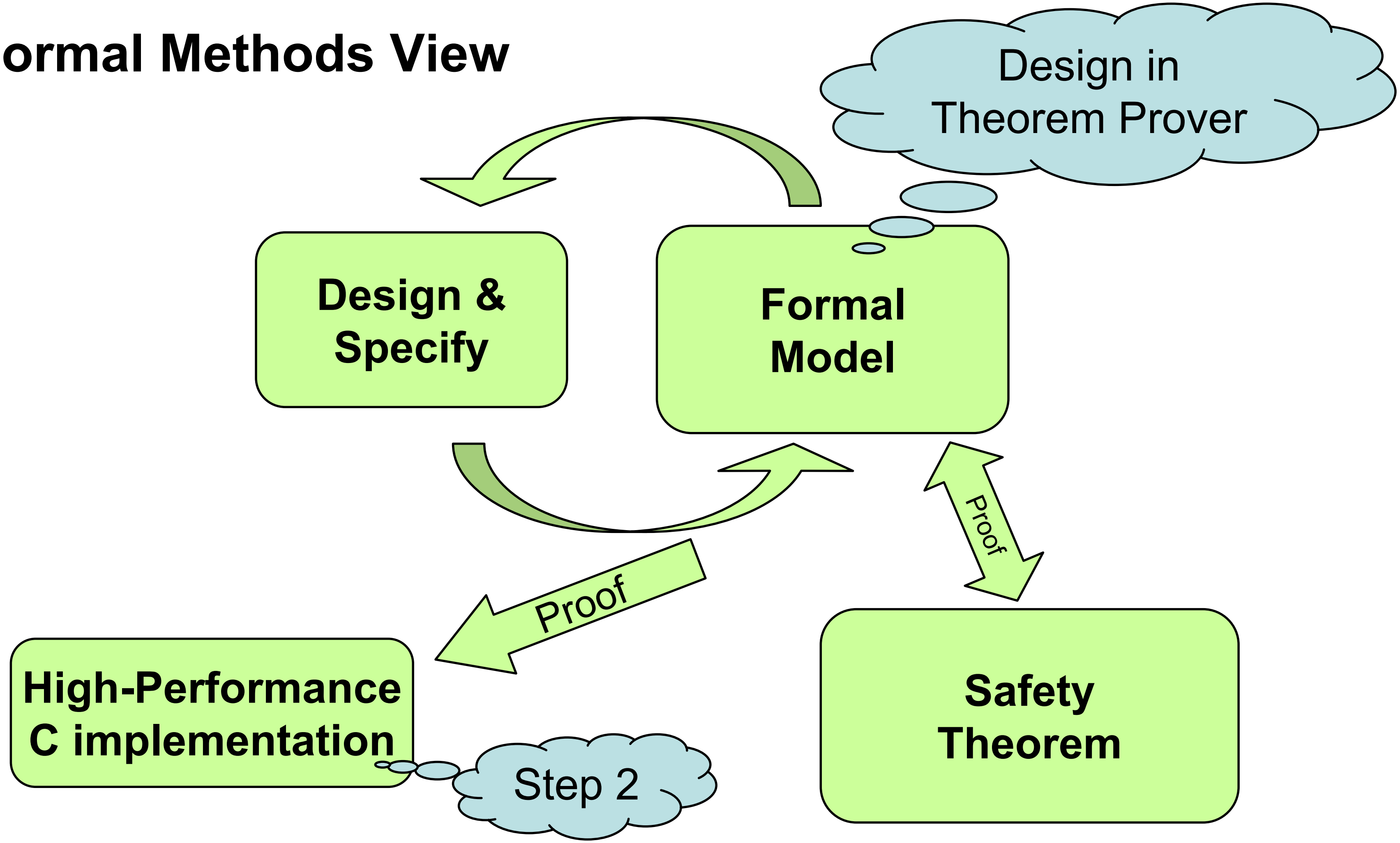
Standard Kernel Design

Kernel Hacker View



Formal Design

Formal Methods View



Design in Theorem Prover

Design & Specify

Formal Model

Proof

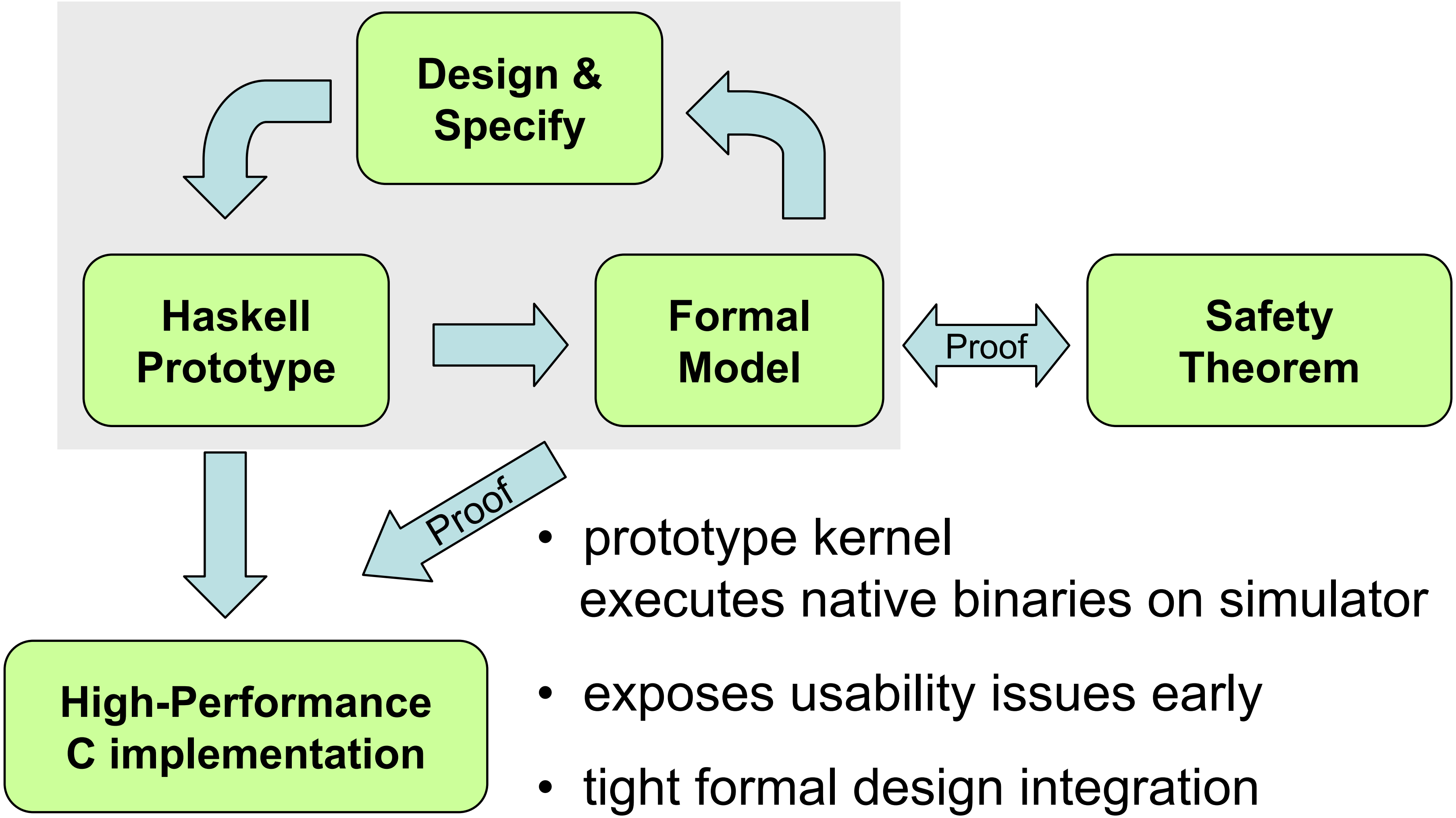
Proof

High-Performance C implementation

Step 2

Safety Theorem

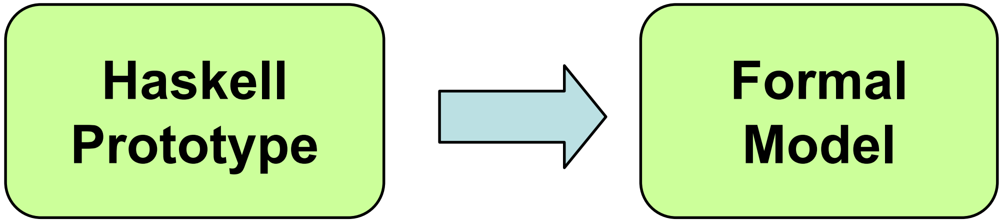
Iterative Design and Formalisation



- prototype kernel
  executes native binaries on simulator
- exposes usability issues early
- tight formal design integration

Haskell to Isabelle/HOL

- Needs to be quick and easy:

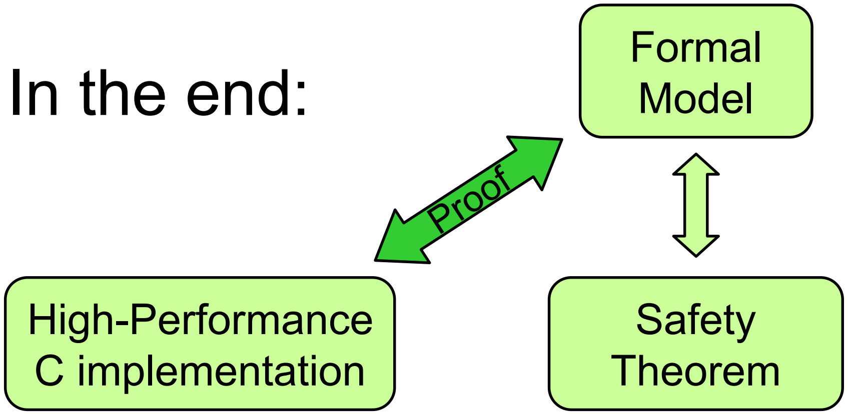


Haskell Prototype → **Formal Model**

- Problems:
  - Size (3000 loc)
  - Real-life code (GHC extensions, no nice formal model)
  - Want Isabelle/HOL for safety and refinement proofs
  - Existing tools do not parse the code

Approach: Quick and Dirty

- In the end:



Formal Model

Proof

High-Performance C implementation

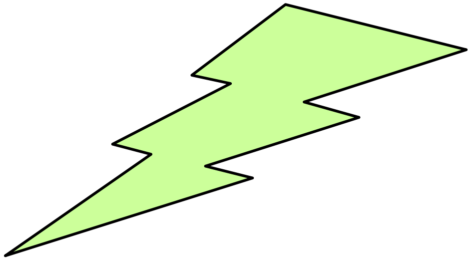
Safety Theorem

- No “hard” translation correctness guarantee
- Remaining issues:
  - Special features (“Dynamic”)
  - Termination
  - Monads

Termination

- **Haskell:**
  - Lazy evaluation
  - Non-terminating recursion possible

- **Isabelle/HOL:**
  - Logic of total functions



Termination

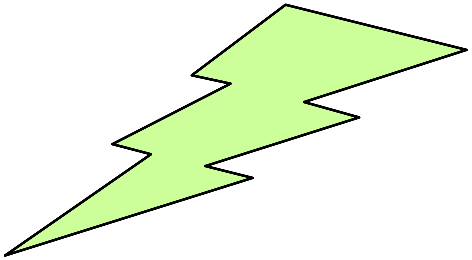
- **Haskell:**
  - Lazy evaluation
  - Non-terminating recursion possible

- **Isabelle/HOL:**
  - Logic of total functions

- **But:**
  - All system calls terminate
  - We prove termination
  - So far: done, relatively easy, not much recursion (cheated once, not really, though)

Monads

- **Haskell kernel:**
  - Imperative, monadic style throughout
- **Isabelle/HOL:**
  - Type system too weak to implement monads in the abstract



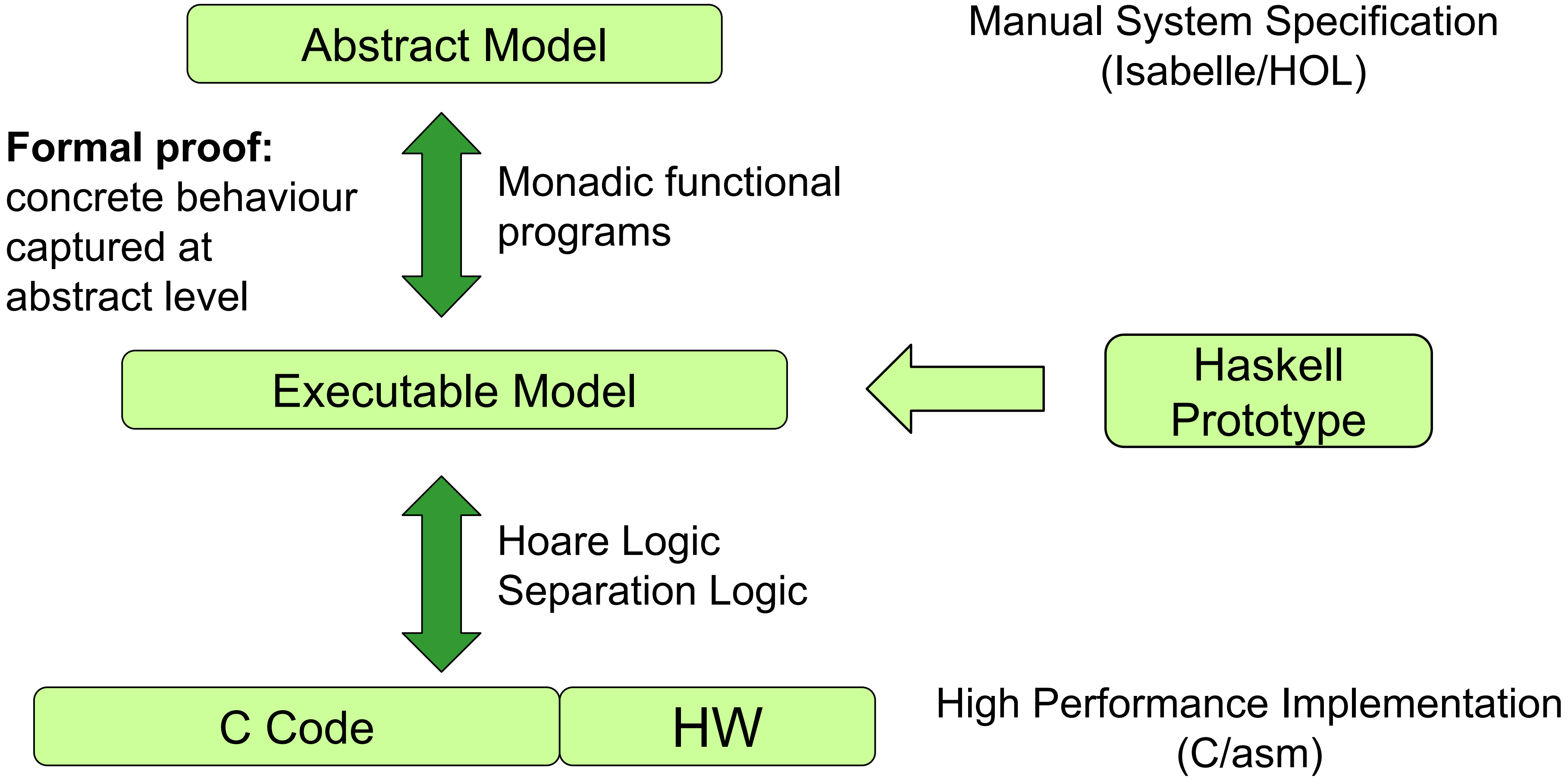
Monads

- **Haskell kernel:**
  - Imperative, monadic style throughout
- **Isabelle/HOL:**
  - Type system too weak to implement monads in the abstract
- **But:**
  - Strong enough to implement concrete monads (state, exception)
  - Nice do-style syntax in theorem prover
  - So far: needed more concrete than abstract properties for proofs

The Proof

Refinement on monadic functional programs

Overview



Abstract Model

Manual System Specification
(Isabelle/HOL)

Formal proof:
concrete behaviour captured at abstract level

Monadic functional programs

Executable Model

Haskell Prototype

Hoare Logic
Separation Logic

C Code

HW

High Performance Implementation
(C/asm)

Refinement

- The old story:
  - C refines A if all behaviors of C are contained in A

- Sufficient: forward simulation

$$\begin{tikzcd}
s \arrow[r, dotted, "A"] \arrow[d, leftrightarrow, "S"'] & t \arrow[d, dotted, leftrightarrow, "S"'] \\
s' \arrow[r, "C"'] & t'
\end{tikzcd}$$

State Monad in Isabelle

- Nondeterministic state monad:

```
types  (σ,α) monad  =  σ ) (α * σ) set

return :: α ) (σ, α) monad
return x s == {(x,s)}

bind (>>=) :: (σ, α) monad ) (α ) (σ, β) monad)
              (σ, β) monad
f >>= g == λs. U (λ(v,t). g v t) ` (f s)

fail :: (σ, α) monad
fail s = {}
```

Hoare Logic for the State Monad

- Hoare triples with result values:

  `{P} f {Q} == ∀s. P s ! (∀(r,s′) ∈ f s. Q r s′)`

- WP-Rules:

$$\frac{}{\texttt{\{P x\} return x \{P\}}}$$

$$\frac{\texttt{\{P\} f \{Q\}} \qquad \texttt{∀x. \{Q x\} g x \{R\}}}{\texttt{\{P\} f >>= g \{R\}}}$$

$$\frac{}{\texttt{\{P\} fail \{Q\}}}$$

State Monad Refinement

- Forward Simulation

$$\begin{tikzcd}
s \arrow[r, dotted, "A"] \arrow[d, leftrightarrow, "S"'] & (r,\ t) \arrow[d, leftrightarrow, dotted, "R"', "S"] \\
s' \arrow[r, "C"'] & (r', t')
\end{tikzcd}$$

```
corres S R A C ==
8(s,s') 2 S.
   8(r', t') 2 C s'.
       9(r, t) 2 A s. (t, t') 2 S Æ (r, r') 2 R
```

State Monad Refinement

- Forward Simulation

$$\begin{tikzcd}
P \arrow[rr, dotted, "A"] \arrow[d, leftrightarrow, "S"'] & & (r,\ t) \arrow[d, dotted, leftrightarrow, "R"'] \arrow[d, dotted, leftrightarrow, shift left=3, "S"] \\
P' \arrow[rr, "C"'] & & (r',\ t')
\end{tikzcd}$$

```
corres S R P P' A C ==
8(s,s') 2 S. P s Æ P' s' ⟶
  8(r', t') 2 C s'.
      9(r, t) 2 A s. (t, t') 2 S Æ (r, r') 2 R
```

A Small Refinement Calculus

$$\frac{}{\texttt{corres S R P P' A fail}}$$

$$\frac{\texttt{(x,y) 2 R}}{\texttt{corres S R P P' (return x) (return y)}}$$

$$\frac{}{\texttt{corres S R P P' (f >>= g) (f' >>= g')}}$$

A Small Refinement Calculus

$$\frac{}{\texttt{corres S R P P' A fail}}$$

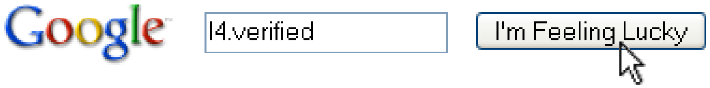
$$\frac{\texttt{(x,y) 2 R}}{\texttt{corres S R P P' (return x) (return y)}}$$

$$\frac{\begin{array}{c}\texttt{corres S R' P P' f f'} \\ \texttt{8x y. (x,y) 2 R'} \longrightarrow \texttt{corres S R (Q x) (Q' y) (g x) (g' y)} \\ \texttt{\{P\} f \{Q\}} \\ \texttt{\{P'\} f' \{Q'\}}\end{array}}{\texttt{corres S R P P' (f >>= g) (f' >>= g')}}$$

Summary

- Monadic style supports Refinement and Hoare Logic nicely
  - get, put, modify, select, or, assert, when, if, case, etc analogous
- Statistics:
  - 3.5kloc abstract, 7kloc concrete spec (about 3k Haskell)
  - 35kloc proof so far (estm. 50kloc final, about 10kloc/month)
  - 22 patches to Haskell kernel, 90 to abstract spec
- Invariants:
  - well typed references, aligned objects
  - thread states and endpoint queues
  - well formed current thread, scheduler queues
- Failure refines everything -> separate proof of "does not fail"

Thank You



l4.verified