Retrofitting Rigour

Peter Sewell

University of Cambridge

July 2014

ITP, Vienna

Retrofitting Rigour

(a hymn to empirical science)

Peter Sewell

University of Cambridge

July 2014

ITP, Vienna

Things I Know About Software

Things I Know About Software

1. there’s a lot of it

Things I Know About Software

1. there’s a lot of it
2. it goes wrong a lot

we’ve failed

we’ve failed

semantics and verification have had regrettably little impact on mainstream system building in the last 50 years

we’ve failed

semantics and verification have had regrettably little impact on mainstream system building in the last 50 years

Perhaps 1 000 000+ s/w and h/w engineers.

0.00. . .01% of s/w is verified functionally correct?

But computer systems are built by

- smart people
- in big groups
- subject to commercial pressures
- using the best components and tools they know....


Sunlight
Little Caesars
HOT-N-READY
PIZZA
HOT-N-READY
POCKETS
POCHETTES
Minute Rice
Beatrice
OASIS
Kraft Dinner
12 PACK
HOT-N-READY
BBQ
pepsi


BANKING
S/W
BROWSER
JS
C
TLS
FILESYSTEM
TCP
ARCH
H/W

This Talk

Before verification (of some component): we have to precisely characterise the existing interfaces

- the assumptions about underlying layers
- the properties it aims to provide

This Talk

Before verification (of some component): we have to precisely characterise the existing interfaces

- the assumptions about underlying layers
- the properties it aims to provide

How to investigate that?

This Talk

Before verification (of some component): we have to precisely characterise the existing interfaces

- the assumptions about underlying layers
- the properties it aims to provide

How to investigate that?

Deeper point: worth doing for its own sake, even without any verification

- each case reveals a fascinating can of worms: deep semantic questions intertwined with fundamental engineering concerns and legacy goop
- an *incremental* way to get semantics (and maybe later verification) into mainstream practice

This Talk

Part 1: experiment and testing

Part 2: consequences for specification tools

This Talk

Examples:

- IBM Power and ARM concurrency (operational)
- ...and axiomatic
- C/C++11 concurrency semantics
- TCP and Sockets API
- PL
- SWIFT: optically switched MAC protocol
- x86 concurrency semantics
- x86, Power, and ARM instruction semantics
- CompCertTSO verified compiler

Current Practice

1. (sometimes, at best) specify in prose
2. write code
3. write some ad hoc tests
4. test-and-fix-and-extend until marketable
5. test-and-fix-and-extend until no longer marketable
6. use until too bitrotted, device breaks, or obsolete

Current Practice

1. (sometimes, at best) specify in prose
2. write code
3. write some ad hoc tests
4. test-and-fix-and-extend until marketable
5. test-and-fix-and-extend until no longer marketable
6. use until too bitrotted, device breaks, or obsolete

Investigating existing abstractions

Sources:

1. from a spec
2. from an existing canonical impl
3. from grey-box testing of existing implementations
4. from discussion with designers/architects
5. from interop testing with existing systems

All of the above, iterated

“Executable” Models?

Executable specifications? Not quite the point.

For this empirical conformance testing between model and implementation, the semantics must be

executable as a test oracle

to decide whether some experimentally observed behaviour is allowed by the model

...and executable as *prosthetic for exploring semantics*

Any nondeterminism or loose specification?

Any nondeterminism or loose specification?

If not, trivial:

- spec can be executable reference impl
- use whatever language is clearest
  (pure functional, algorithmic inductive relation, C,...)
- run on same input, check equal output

Any nondeterminism or loose specification?

If so, more interesting. Tied in with:

- mathematical form of spec
- size and nature of test cases
- observability of implementations
- social/legal context

Look at examples

Operational h/w memory models (Power/ARM)

[Sarkar,Maranget,Alglave,Williams,Sewell]

Existing implementations? Yes – something to test

Operational h/w memory models

Existing implementations? and they really differ:

		POWER			ARM	
	Kind	PowerG5	Power6	Power7	Tegra2	
MP	Allow	10M/4.9G	6.5M/29G	1.7G/167G	40M/3.8G	
MP+dmb/sync+po	Allow	670k/2.4G	0/26G U	13M/39G	3.1M/3.9G	
MP+dmb/sync+addr	Forbid	0/6.9G	0/40G	0/252G	0/29G	
MP+dmb/sync+ctrl	Allow	363k/5.5G	0/43G U	27M/167G	5.7M/3.9G	
MP+dmb/sync+ctrlsib/isync	Forbid	0/6.9G	0/40G	0/252G	0/29G	
S+dmb/sync+po	Allow	0/2.4G U	0/18G U	0/35G U	271k/4.0G	

Operational h/w memory models

Existing implementations? form some kind of de facto standard – that's what codebase is tested against

Operational h/w memory models

Existing implementations? form some kind of de facto standard – that's what codebase is tested against

But architecture texts deliberately looser

Operational h/w memory models

Existing implementations? form some kind of de facto standard – that's what codebase is tested against

But architecture texts deliberately looser

Also some processors don't conform

Operational h/w memory models

Observability of implementations

Experimental
opaque execution – just final register and memory state

but (empirically) relatively good exploration of set of executions – Luc Maranget's litmus magic, $\sim 10^{10}$ iterations/test

(opposite situation for pre-silicon testing)

Operational h/w memory models

Observability of implementations

Test generation [Alglave,Maranget] Model inspired, not model-generated

enumerate non-SC cycles (`Rfe PodRR Fre DMBdWW`) and generate:

```
ARM MP+dmb+po
{ %x0=x; %y0=y; %y1=y; %x1=x; }
 P0                    | P1
 MOV R0, #1            | LDR R0, [%y1]
 STR R0, [%x0]         | LDR R1, [%x1]
 DMB                   |
 MOV R1, #1            |
 STR R1, [%y0]         |

exists (1:R0=1 and 1:R1=0)
```

Operational h/w memory models

Observability of implementations

Legal and Social
Opaque designs — so black-box testing

But can talk with architects and designers to limited extent:

- intuition for internal structure
- judgment calls on architectural intent

(IBM, ARM, Qualcomm, AMD)

Operational h/w memory models

Mathematical form of semantics

- unlabelled transition system $s \longrightarrow s'$
- observation function obs from final s to register and memory state
- massively nondeterministic (cf ppcmem)

Operational h/w memory models

Mathematical form of semantics

- unlabelled transition system $s \longrightarrow s'$
- observation function obs from final s to register and memory state
- massively nondeterministic (cf ppcmem)

But interesting examples are small, so can compute

$$\{obs(s) | s_0 \longrightarrow^* s \not\longrightarrow\}$$

by exhaustive search

and check set inclusion wrt experimental observations

Can also explore single paths interactively (or heuristically)

Operational h/w memory models

Mathematical form of semantics

- unlabelled transition system $s \longrightarrow s'$
- observation function obs from final s to register and memory state
- massively nondeterministic (cf ppcmem)

Intensional structure matters

$$s = s_{\text{storage}} \mid s_{\text{thread 1}} \mid \ldots \mid s_{\text{thread N}}$$

abstracting microarchitecture in architect-friendly way (intensional and intentional validation...)

c.f. sync acks and DMB

Operational h/w memory models

Inventing precise architectural abstractions

incrementally...

Axiomatic h/w memory models

[Alglave,Maranget,Sela Mador-Haim,...]

Same experimental context

Different mathematical form of semantics

1. Threadwise semantics

   $$\textit{program} \mapsto \textit{set of candidate complete executions}$$

   Each corresponds to a control-flow unfolding and an arbitrary choice of memory read values

   Inductive on program structure, “routine”

2. then filter with predicate on candidate execution (acyclicity of various relations,...)

Axiomatic h/w memory models

Algorithmically: can find all executions of small examples. Still exponential, but rather faster in practice than (intentionally naive) path exploration

$$\{obs_{ax}(x_t, x_w) | x_t \in \mathrm{threadwise}(prog) \wedge \\ x_w \in \mathrm{all\_rf\_co}(x_t) \wedge \\ \mathrm{consistent}(x_t, x_w)\}$$

But can't explore single paths (e.g. to walk along emulation trace)

Simpler for some proofs; harder for others (induction?)

Harder to relate to microarch

Aside: Testing Metatheory Too

For equivalence between operational and axiomatic models:

- compare allowed outcomes on same test suite (9 000 litmus tests)

Aside: Testing Metatheory Too

For equivalence between operational and axiomatic models:

- compare allowed outcomes on same test suite (9 000 litmus tests)
- make mappings between operational and axiomatic executions executable and test

if setup right, easy first step before (and complement to) hand or mechanised proof

C/C++11 axiomatic PL memory model

[Batty, Owens, Sarkar, Sewell,...]

behaviour emerges from compiler + hardware

Existing implementations? Not during design

Now supported by GCC and Clang — code open but inscrutable

Hard to restrict existing optimisations, but compilation of atomics mutable

C/C++11 axiomatic PL memory model

Mathematical form of semantics

1. threadwise semantics produces set of candidate complete executions
2. filter with consistency predicate
3. if any consistent execution is racy, program is undefined

(far from traditional opsem or densem)

Also massively nondeterministic

Again can enumerate exhaustively for litmus test examples, but combinatorics gets tricky quickly (cppmem)

(Equivalent operational version by Nienhuis — out-of-order and symbolic)

C/C++11 axiomatic PL memory model

Experimental testing
much harder:

- much bigger examples needed to trigger compiler optimisation
  (much too big to run semantics on them exhaustively)
- harder to explore set of executions

C/C++11 axiomatic PL memory model

Experimental testing
much harder:

- much bigger examples needed to trigger compiler optimisation
  (much too big to run semantics on them exhaustively)
- harder to explore set of executions

but devious scheme of [Morisset, Pawan, Zappa Nardelli] avoids reliance on randomness:

- experimental observability of single-thread memory trace, by binary rewriting
- prove facts about what changes are legal; use those rather than evaluating semantics directly

TCP

[Bishop, Norrish, Ridge, Sewell, Wansborough,...]

Existing implementations? Yes - and they are de facto standard

Observability of implementations

- **Experimental** visible wire, Sockets API, and debug events
  Need medium-length traces to explore behaviour, and test harness to fake wire events
- **Test generation** semi-systematic of 6000 traces
- **Legal and social** BSD, Linux open; Windows closed. Code semi-scrutable – some reverse engineering of spec

TCP

Mathematical form of semantics

- labelled transition system $s \xrightarrow{l} s'$ labelled with wire, API, debug events, and *time passage*
- lots of nondeterminism, in implementations and in spec — and much is *internal*

Given experimental trace $l_1, \ldots, l_n$, try to decide whether admitted ($\exists s.\ s_0 \rightarrow^* \xrightarrow{l_1} \rightarrow^* \xrightarrow{l_2} \ldots \rightarrow^* \xrightarrow{l_n} s$)

- maintain symbolic state as HOL4 formula
- simplify
- backtrack occasionally

Heavy!

TCP

Conformance testing along trace is highly discriminating:

- The received urgent pointer is not updated in the fast-path code, so if 2GB of data is received in the fast path, subsequent urgent data will not be correctly signalled.
- After 2^{32} segments there is a 16 segment window during which, if the TCP connection is closed, the RTT values will not be cached in the routing table.
- The receive window is updated on receipt of a bad segment.
- Simultaneous open can respond with an ACK rather than a SYN,ACK.
- The code has an erroneous definition of the `TCPS_HAVERCVDFIN` macro, making it possible, for example, to generate a `SIGURG` signal from a socket after its connection has been closed.
- `listen()` can be (erroneously) called from any state, which can generate pathological segments (with no flags or only a FIN).

Moral

minimise internal nondeterminism

(by protocol/API design and test harness instrumentation)

make the standards testable (executable as test oracle)

Programming languages

bizarrely, almost no testing of relationship between semantics and implementations!

or of basic properties, eg type preservation

Impact

many pre-verification benefits

- identify key examples and phenomena
- establish de facto standards
- fix industry specs
- build exploration tools
- provide test suites
- ...

(all things that practitioners can readily engage with)

plus base semantics for verification

Part 2: writing specifications

Need *reusable* models

Establishing (justified, validated, accepted) models can be a big undertaking.

Must be generally reusable, by many groups for many purposes.

Lem

[Owens, Mulligan, Tuerk, Gray, Bohm, Zappa Nardelli, Sewell,...]

Language and tool for engineering such specifications

System available; rough diamond in ITP 2011; paper at ICFP 2014

Discuss requirements, design, experience
(c.f. prover front-ends)

Moderately large-scale definitions

- Power/ARM operational concurrency model: 3000 LoS
- Power/ARM axiomatic concurrency model: 1100 LoS
- C/C++11 axiomatic concurrency model: 1500 LoS
- TCP and Sockets API: 6700 LoS

- $\text{OCaml}_{\text{light}}$ semantics: 3100 LoS (Owens, from Ott)
- CakeML semantics, compiler: 4900 LoS (Kumar et al.)
- C Core semantics: 8200 LoS (Memarian)
- C/C++11 operational concurrency: 1100 LoS (Nienhuis)
- Operational no-thin-air model: 1100 LoS (Pichon)

Modest mathematical demands

FP features:

- pure higher-order functions (plus monads, sometimes)
- recursive definitions
- recursive algebraic datatypes, lists, records
- pattern matching
- top-level ML parametric polymorphism
- simple type classes for overloading
- simple module system

Logic and sets:

- universal and existential quantification (higher-order)
- sets and set comprehensions, relations, finite maps
- inductive relation definitions
- lemma statements

Modest mathematical demands

No variable binding

No dependent types (or very simple ones)

No type abstraction

No functors

...these things are behaviourally subtle, but their whole-system models can be mathematically straightforward

Tool requirement 1: Executability as Test Oracle

Everything is finite (in executions of small test cases) — quantify only over concrete finite sets

Power/ARM operational model: computable transition preconditions and resulting states

Power/ARM and C/C++11 axiomatic models: exhaustively enumerate candidate complete executions, computable predicates over those

C/C++11 operational models: need some symbolic execution

TCP: needed very symbolic execution (HOL4, not Lem)

Tool requirement 1: Executability as Test Oracle

System integration:

need executable code in conventional PL that can be linked with front-end test parser, memoising exploration, and GUI

Lem $\xrightarrow{\texttt{lem}}$ OCaml $\xrightarrow{\texttt{ocamlopt}}$ Batch executable

Lem $\xrightarrow{\texttt{lem}}$ OCaml $\xrightarrow{\texttt{ocamlc}}$ OCaml bytecode $\xrightarrow{\texttt{js_of_ocaml}}$JavaScript

Tool requirement 1: Executability as Test Oracle

Performance?

sometimes can be (almost) completely naive

sometimes mildly tweak model or execution harness

state spaces on the limit of viability, and symbolic is worse (cluster for ppcmem, 100 machines for weeks for TCP...)

Tool requirement 2: Presentation

We have to quote definition fragments in multiple documents.

Scale, intricacy, and lifetime means this must be *automatic*

To be comprehensible, need decent typesetting and *manual control of layout*

Tool requirement 2: Presentation

```
let visible_side_effect_set actions hb =
  { (a,b) | forall ((a,b) IN hb) |
    is_write a && is_read b && (loc_of a = loc_of b) &&
    not ( exists (c IN actions). not (c IN {a;b}) &&
        is_write c && (loc_of c = loc_of b) &&
        (a,c) IN hb && (c,b) IN hb) }
```

Tool requirement 2: Presentation

```
let visible_side_effect_set actions hb =
  { (a,b) | forall ((a,b) IN hb) |
    is_write a && is_read b && (loc_of a = loc_of b) &&
    not ( exists (c IN actions). not (c IN {a;b}) &&
        is_write c && (loc_of c = loc_of b) &&
        (a,c) IN hb && (c,b) IN hb) }
```

To use Lem-typeset version:

```
\LEMvisibleSideEffectSet
```

from

```
alldoc-inc.tex
```

Tool requirement 2: Presentation

```
let visible_side_effect_set actions hb =
  { (a,b) | forall ((a,b) IN hb) |
    is_write a && is_read b && (loc_of a = loc_of b) &&
    not ( exists (c IN actions). not (c IN {a;b}) &&
        is_write c && (loc_of c = loc_of b) &&
        (a,c) IN hb && (c,b) IN hb) }
```

To use Lem-typeset version:

let $visible\_side\_effect\_set\ actions\ hb\ =$

$\{\ (a,\ b)\ |\ \forall\ (a,\ b) \in hb\ |$

is_write $a \wedge$ is_read $b \wedge$ (loc_of a = loc_of b) $\wedge$

$\neg\ (\ \exists\ c \in actions.\ \neg\ (c \in \{a,\ b\}) \wedge$

is_write $c \wedge$ (loc_of c = loc_of b) $\wedge$

$(a,\ c) \in hb \wedge (c,\ b) \in hb)\ \}$

Tool requirement 2: Presentation

And linkable HTML

———

And automatic production of execution graphs

And hand-translations into English

And for TCP more elaborate special-purpose structuring

Tool requirement 3: Simplicity

- simple usage model:

  must fit into more complex build process and workflow:
  - command-line tool (compiler-style)
  - typechecks definitions
  - generates executable code, typesetting, prover defns
- clean language design
- decent library for specification
- scales to large specs (1–10k LOS)

Make writing specifications more like programming
(accessible to FP-friendly systems people)

Tool requirement 4: usable prover definitions

Routinely need to make models available in multiple provers (HOL4, Isabelle/HOL, Coq). Have to avoid *prover lock-in*.

Need *usable source definitions*, not opaque lambda terms.

In general v. hard, but here in simple fragment. Doable?

Previous state of the art for porting definitions: `sed`

Tool requirement 4: usable prover definitions

Whitespace

Where possible, preserve source whitespace layout and comments

Try to generate readable idiomatic code.

Tool requirement 4: usable prover definitions

Target-specific encodings

Encode around annoying target differences, eg pattern matching:

	OCaml	Coq	HOL4	Isabelle/HOL
record patterns	y	y	n	n
as-patterns	y	y	n	n
zero-and-succ	n	y	y	y
arb. patterns in lets	y	n	n	n
non-exhaustive	y	n	y	y
redundant rows	y	n	some	n

...minimal pattern-match compilation.

Tool requirement 4: usable prover definitions

Equality

Isabelle/HOL and HOL4 have pervasive boolean equality constant $\alpha \to \alpha \to \mathsf{BOOL}$

OCaml has pervasive equality constant safe to use at some types

Coq's equality has type $\alpha \to \alpha \to \mathsf{PROP}$

...use Lem's type-class machinery, but with mechanism to suppress dictionary-passing for HOL4 and Isabelle/HOL

Tool requirement 4: usable prover definitions

Target-specific bindings

Sophisticated binding and renaming mechanism

Lem understands backend namespaces, automatically renames constants

Can e.g. change order of arguments to a bound function in a backend, inline a definition completely, rename functions, etc.

Tool requirement 4: usable prover definitions

Lem library

Functions, relations, sets, lists, finite maps, machine words etc.

Partitioned:

- standard library portable in all backends
- ‘extra’ may only be in some, or be underspecified

`nat` and `natural` etc.

Tool requirement 4: usable prover definitions

Meaning preservation?

Lem Implementation

[Owens, Mulligan, Tuerk, Gray, Bohm, Zappa Nardelli, Sewell,...]

Tool typechecks Lem definitions and generates

- LaTeX and (simple) HTML
- OCaml
- HOL4, Isabelle/HOL, and (preliminary) Coq

Implementation in 29 000 lines OCaml, 7 700 of Lem libraries, 3 800 of target libraries. 3.5 person-years.

Tool available. Paper in ICFP 2014.

Lem in use

- Operational concurrency model for Power and ARM (3000 LoS, Sarkar et al)
- Axiomatic concurrency model for C/C++11 (1500 LoS, Batty et al)
- Correctness proofs for compilation scheme from C/C++11 concurrency primitives to Power (900 LoS for lemmas, Batty et al)
- Axiomatic memory model for Power (1100 LoS, Mador-Haim et al)
- TCP/IP protocols and Sockets API specification (6700 LoS, Bishop et al. ported from HOL4
- $\text{OCaml}_{\text{light}}$ specification (3100 LoS, Owens et al, generated from Ott)
- CakeML language semantics and verified compiler (4900 LoS, Kumar et al.)
- C Core semantics (8200 LoS, Memarian)
- Operational concurrency model for C/C++11 (1100 LoS, Nienhuis)
- Operational no-thin-air model (1100 LoS, Pichon)

Lem and Ott

Ott: earlier tool [Sewell, Zappa Nardelli, Owens]

- user-defined syntax
- inductive relations
- generate LaTeX, and provers (Coq, HOL4, Isabelle), not execution

Lem:

- FP-style fixed syntax
- types, functions, and inductive relations
- generate LaTeX, OCaml, and provers (Coq-ish, HOL4, Isabelle)

Complementary — and can generate Lem from Ott.

Lem Status

It’s great

Lem Status

It's great

except:

- Lem's type classes are a bit too simple
- really idiomatic?
  (esp. to generate *good* Coq code, need refined typing)
- inductive relation code generation not fully implemented
- want random test generation
- internals could do with reengineering
- really want Lem+Ott upper bound, including parser/pp gen.

Conclusion

- investigate behaviour of extant abstractions
- ...as foundation for verification and for its own sake
- with combination of experiment, white-box investigation, testing, and engagement with designers.

Key technical point: semantics must be

executable as a test oracle

Conclusion

- investigate behaviour of extant abstractions
- ...as foundation for verification and for its own sake
- with combination of experiment, white-box investigation, testing, and engagement with designers.

Key technical point: semantics must be

executable as a test oracle

- what language and tool support do we need for large-scale specification?
- Lem, as a step towards that

Path to incremental engagement with mainstream practice?