Netsem, Ott, Lem, SibylFS and Verified Parsing: Specification and Validation at Scale

Dr Tom Ridge

2016-04-20

Topics

Talk will include

- Description of the projects in the title, and some interesting aspects of each
- General comments about specification, validation and proof for real-world systems
- Because I know some people are also interested in formalized proof systems, I will talk a bit about work I did in this area (mostly unpublished)

Outline of talk

Roughly, I will present things in chronological order

- Early work
- Netsem (a large, formal specification of UDP,TCP/IP and ICMP)
- POPLmark (can theorem provers handle large formalizations?)
- Ott and Lem (tools for large formalizations)
- SibylFS (a large, formal specification of file systems)
- Current/future work

Early stuff

- There are some theories in the Archive of Formal Proofs e.g. a version of Ramsey's theorem (read Boolos and Jeffrey C+L!)
- Soundness and completeness for FOL (with Margetson - he did most of the work), with an executable (within Isabelle, or via extracted code), complete proof search for FOL
  - This is based on Wainer and Wallen BPT
  - N.B. for FOL proof search, resolution is probably "better" than tableaux (?)
- Formalized proof theory e.g. Craig's interpolation theorem (unpublished)
  - Based on Girard PT+LC
- PhD: applied formal methods

c. 2005

- Motivation: I really want to improve the systems we currently use
- Observation: formalization (+ mechanization) is an extremely effective way of developing correct software (really, the only way)
- Observation: automation is important, but for non-trivial systems you need interactive proof
- Observation: (as any fule kno) F+M takes a long time

Why does it take so long?

- Lots of details; "obvious" results can be painful to get the machine to understand
- Logics are inflexible in various ways
- Tools are immature
- Formalized libraries are quite small still; often they involve shortcuts which limit their reuse
- Proofs
  - Existing "proofs" are not proofs at all (at best, optimistic; some notable exceptions)
  - Existing proofs are unsuitable for mechanization (even if they are roughly correct)
  - So, you effectively have to come up with your own proofs, and this takes time; it also requires formalizers to be fairly bright, and so the talent pool is fairly small, which matters
- Community obsessed with formalizing everything (without assumptions), which makes progress very slow
  - Why not have a formal repository of defns and theorem statements? Because we haven't got the right logics/formalisms

So what next?

- Apply F+M to real-world systems; ideally produce systems that are better than existing systems (features, performance,...), and proved correct
- Try to solve the engineering problems (reduce the effort)
- Try to see what the other problems are
- Take a long-term view
- Focus on the components and interfaces that are most "valuable"

Outline of talk

- Early work
- **Netsem**
- POPLmark
- Ott and Lem
- SibylFS
- Current/future work

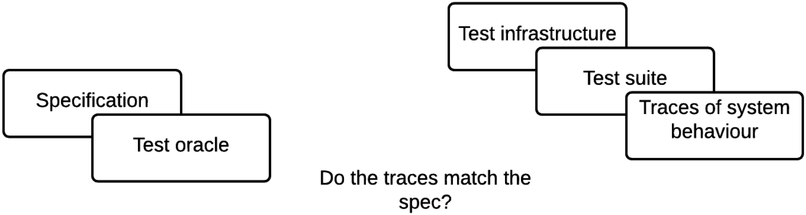
First big project: Netsem

- 2001-c.2009 (some work still ongoing), Peter Sewell et al., http://www.cl.cam.ac.uk/~pes20/Netsem/
- "apply formal methods to the network stack"
- develop a spec of UDP+TCP/IP+ICMP
- test spec against real-world implementations - to validate spec, and discover bugs in implementations

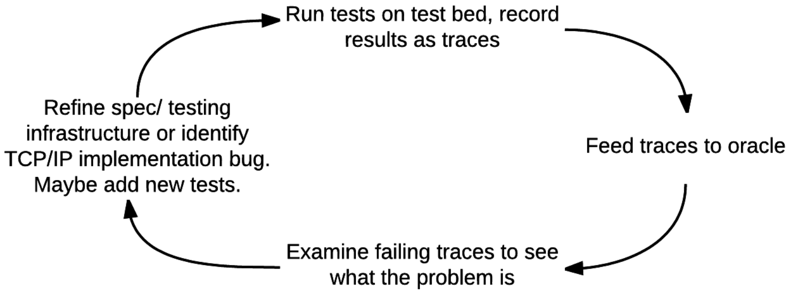
Netsem TCP/IP specification

- Concurrency, distribution clearly important areas of computer science, particularly with the arrival of the internet
- Academia: pi calculus etc; my uninformed view: very useful, but no real attempt to address e.g. host and network failure
- Real-world network programming uses TCP/IP, and has to take these issues seriously; particularly timers, timeouts etc

Netsem TCP/IP specification, artefacts



Netsem workflow



Netsem TCP/IP specification

- Written in HOL (HOL4), with various specification "idioms" ("relational monad")
- Relatively close correspondence to BSD implementation code (probably a mistake)
- Extensively documented
- Available at http://www.cl.cam.ac.uk/~pes20/Netsem/ and now on GitHub https://github.com/PeterSewell/netsem

Spec excerpt

recv_1 **tcp: fast succeed** **Successfully return data from the socket without blocking**

$$
\begin{array}{l}
h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d);\\
\quad socks := socks \oplus\\
\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,\\
\qquad\qquad \mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq, rcvurp, iobc)))]|\!\rangle\\
\xrightarrow{tid\cdot\mathrm{recv}(fd, n_0, opts_0)}\ h\ \langle\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\mathrm{OK}(\textbf{implode}\ str, *)))_{\mathrm{sched\_timer}});\\
\quad socks := socks \oplus\\
\qquad [(sid, \textsc{Sock}(\uparrow fid, sf, is_1, ps_1, is_2, ps_2, es, cantsndmore, cantrcvmore,\\
\qquad\qquad \mathrm{TCP\_Sock}(st, cb, *, sndq, sndurp, rcvq'', rcvurp', iobc)))]|\!\rangle
\end{array}
$$

$$
\begin{array}{l}
((st \in \{\mathrm{ESTABLISHED}; \mathrm{FIN\_WAIT\_1}; \mathrm{FIN\_WAIT\_2}; \mathrm{CLOSING};\\
\quad \mathrm{TIME\_WAIT}; \mathrm{CLOSE\_WAIT}; \mathrm{LAST\_ACK}\} \wedge
\end{array}
$$

This describes a host transition. This is lifted to the "network" level to give transitions of a network of machines.

Test infrastructure and oracle

- Written in SML, OCaml, C, and HOL (with perl, bash etc)
- Infrastructure to run distributed test cases (multiple hosts communicating over network) and record (distributed) traces. Logging of events on wire, in TCP/IP stack and at sockets API on every host, and merged to form a single global trace
- Oracle, to check traces against the spec

Formal foundation: labelled transition systems

- (S, L, S_0, T) where $S_0 \subseteq S$; $T \subseteq S \times L \times S$
- Notation: $s \xrightarrow{l} s'$ means $(s, l, s') \in T$
- T defined by clauses of the form: $P(s, l, s') \longrightarrow (s \xrightarrow{l} s')$
- A trace is $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ldots$. The spec tells you which of these are "OK" and which not.
- Each state $s \in S$ describes a **network** of machines, and the TCP/IP, UDP and ICMP packets on the wires
- Observation: LTS/small-step operational semantics is a very basic and general formalism; it is usually my go-to formalism

Example trace

https://github.com/PeterSewell/netsem/blob/master/demo-traces/trace1741#L87-L112

Or here

Problem (partial states and labels, partial traces)

A trace is $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ldots$

- The event logging infrastructure observed partial details of the states s_i
- And partial details of the labels/events l_i
- And some events were not observable (internal kernel transitions, τ events)

So the checker was given information like:
$s_0 \xrightarrow{l_1} s_1(partial) \xrightarrow{?} s_2 (\xrightarrow{?})?\ s_? (\xrightarrow{?})?\ s_i \ldots$
And had to figure out the complete trace as above.

- When searching for such a trace, the nondeterminism is huge
- The test oracle was implemented in SML, using the symbolic execution engine of HOL4, and the HOL4 specification (so guaranteed sound, but slow)

Result: checking traces was very very slow e.g. 2500 hours to check 1000 traces

Possible solution: logging infrastructure should record everything

1. $s_0 \xrightarrow{l_1} s_1(\mathit{partial}) \xrightarrow{?} s_2 \; (\xrightarrow{?})? \; s_? \; (\xrightarrow{?})? \; s_i \ldots$
2. $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ldots$

We don't want (1), we do want (2). This is maybe possible with current kernel tracing technology, but likely that parts of the s_i are still partial/ symbolic.

Problem: spec not suitable for checking traces

- Suppose we have $s \stackrel{l}{\rightarrow} s'$ where we have full information about s, l, s'.
- The specification is given in the form $P(s, l, s') \longrightarrow (s \stackrel{l}{\rightarrow} s')$
- This is a **specification**. There is no guarantee that it is possible to compute whether $P(s, l, s')$ is satisfied or not.
- For Netsem, we wrote transformations to transform the spec into a form that could be executably checked, but this was ad hoc and painful.

Other problems

- Spec large and unwieldy (partly because it stuck closely to BSD code structure)
- Hard to debug spec (partly because it stuck closely to BSD code structure, partly because of a lack of tools, partly because things took so long)

Netsem was painful; I don't want to do that again

Outline of talk

- Early work
- Netsem
- **POPLmark**
- Ott and Lem
- SibylFS
- Current/future work

Detour: Mechanized Metatheory for the Masses: The POPLmark Challenge

- Sewell et al., 2005, "Mechanized Metatheory for the Masses: The POPLmark Challenge"
- Are theorem provers up to the task of mechanizing PL research? (Really: are TPs up to the task of mechanizing certain areas of CS research which involve intricate syntax-heavy proofs)
- Case study: System F-sub
- Quite a few solutions submitted (about 10)
- Current status
  - Pierce et al., 2008, "It Is Time to Mechanize Programming Language Metatheory"
  - 2015: Most (?) POPL papers are accompanied by mechanized artefacts

Unpublished work

- I did the POPLmark challenge in HOL4, using a named representation and explicit alpha variance in lemma statements when needed
- I focused on making the proofs human readable, and my feeling is that they are very nice proofs
- Overall the attempt was almost successful, but there was one point where the proof really needed equality of alpha-equivalent terms, and stating the lemmas using explicit alpha was just too painful.
- An alternative would be to induct on “sizes” of terms and derivations, but of course this makes the proofs slightly less nice.

Thoughts

- All the major theorem provers can handle this sort of case study
- For me, De Bruijn is clearly the most successful representation
  - Reuse is potentially a problem because the same lemmas have to be proved for each new datatype-with-binding; machinery helps here, or moving to a more general setting (e.g. set theory)
- With the exception of De Bruijn representations, binding is problematic because of the desire for equality of alpha-equivalent terms
  - And theorem provers such as HOL4 are very bad at quotient types (is Coq better here? probably not)
- So maybe use De Bruijn; if De Bruijn is not suitable (e.g. lots of datatypes-with-binding), one might consider Pitts-style nominal approaches via Nominal Isabelle (because the machinery proves a lot of lemmas for you)

Also around this time, some more unpublished work

- Type soundness for various systems
- Some attempts at formalizing cut elimination, Craig's interpolation theorem etc
- Many attempts at formalizing various approaches to binding
- Some comments in a minor workshop "Workshop on Mechanizing Metatheory" http://www.tom-ridge.com/resources/doc/ridge07wmm.pdf

Outline of talk

- Early work
- Netsem
- POPLmark
- **Ott and Lem**
- SibylFS
- Current/future work

What is Ott?

- Ott, c. 2006-? http://www.cl.cam.ac.uk/~pes20/ott/
- "Ott is a tool for writing definitions of programming languages and calculi"
- Write a definition in a `.ott` file, and get output in tex, Isabelle, Coq, HOL4
- Disclaimer: I was a user, not a developer

Example

```
metavar termvar, x ::=   {{ com  term variable }}
{{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-equalit
{{ ocaml int}} {{ lex alphanum}} {{ tex \mathit{[[termvar]]

grammar
t :: 't_' ::=                                    {{ com term
  | x             ::  :: Var                       {{ com variab
  | \ x . t       ::  :: Lam (+ bind x in t +) {{ com lambda
  | t t'          ::  :: App                       {{ com app
  | ( t )         :: S:: Paren                     {{ icho [[t]]
  | { t / x } t' :: M:: Tsub
                          {{ icho (tsubst_t [[t]] [[x]] [[t']
```

Example

Some more declarations, then . . .

Example

```
---------------------------  :: ax_app
(\x.t12) v2 -->  {v2/x}t12

t1 --> t1'
-------------- :: ctx_app_fun
t1 t --> t1' t

t1 --> t1'
-------------- :: ctx_app_arg
v t1 --> v t1'
```

OCaml light (major use of Ott)

- See http://www.cl.cam.ac.uk/~so294/ocaml/
- 67 pages of typeset defns for the core language (no modules, no objects etc)

Why not use a theorem prover?

- TP-specific definitions - no reuse across TPs
- Usually heavier syntax in a TP
- TPs are slow; Ott is lightweight and quick to execute and performs various forms of syntax and type-checking - perfect for e.g. writing a paper (cf. Isabelle, where big developments can take hours to process)

Ott: current status

- According to google scholar, 91+88 citations
- I don't know for sure, but I expect that it is currently used by maybe 10 or so researchers (but is this "good" or "bad"?)
- I think they are mostly fairly happy with it
- e.g. Harley Eades 2014 thesis "The semantic analysis of advanced programming languages": every language in the thesis formalized with Ott; "Ott is a great example of a tool using the very theory we are presenting in this thesis"
  - "In addition, the full Ott specification of every type theory defined with in this thesis can be found in the appendix"
  - This is great. I really want CS and Maths to be presented using formal **definitions**, even if the proofs are informal

Possible alternative to Ott

- A lot of Ott is parsing/pretty-printing, but there is a whole lot of other stuff in there as well (eg type checking)
- Thought: good parsing/pretty-printing technology could get a long way towards Ott feature set, with little of the cost
  - Write definitions
  - Use parser/printer to output to some other system (e.g. OCaml or Isabelle)
  - Use type checking of the other system

A sideline: parsing

Parsing

- During Netsem, I had the opportunity to use Lex and Yacc
- I wanted something else: a general parser, using combinators, that was fast and easy to use, and based on a "good" theoretical foundation
- The problem with naive combinator parsers
  - "left recursion" , e.g. `A -> A B`
  - "epsilon productions", i.e. `A -> ""`

Verified combinator parsing

- c.2010-2011
- Combinator parsing for all CFGs
- Sound and complete
- Verified in HOL4
- Introduced some new concepts ("good" parse trees)
- As a by-product, provided a proof of correctness for a line of work by Frost et al.
- But unusably slow in certain situations

More parsing

- I wanted something that was "fast", and even more flexible (more flexible than arbitrary CFGs? yes)
- This "verified parsing" work was useful because it forced me to learn about algorithms and complexity
- More recent approach "Simple, efficient, sound and complete combinator parsing for all context-free grammars, using an oracle." (Ridge, SLE 2014)
  - Use combinators for the interface, but use Earley to do the actual parsing
  - Introduced the idea of representing parse information using an oracle, not a "shared, packed, parse forest"
- The parsing work continues; essentially I have extended the combinator/Earley approach to treat infinite CFGs, and mildly-context-sensitive grammars (e.g. to parse indentation-sensitive languages like Markdown); unpublished

Lem

Lem, 2011-c. today

- "Lem is a lightweight tool for writing, managing, and publishing large scale semantic definitions."
- cf. Ott, Lem is not specialized to PL defns: "Lem is a general-purpose specification language, whereas Ott is a domain-specific language for writing specifications of programming languages (i.e., inductive relations over syntax)."
- Again, write definitions etc in a `.lem` file and export to Isabelle, HOL4, Coq, OCaml, tex
- Netsem definitions ported to Lem from original HOL4
  - using some fancy parsing machinery that I was working on to parse the defns from HOL4 and pretty-print them to Lem
  - amusingly, the stricter Lem syntax caught a bug in the spec
- Lots of other large case studies use Lem e.g. weak memory models, CakeML etc etc
- Disclaimer: I am a user not a developer

Lem example

```
val is_root_dir : forall 'dir_ref 'file_ref 'impl.
  fs_ops 'dir_ref 'file_ref 'impl ->
  'impl ->
  'dir_ref ->
  bool
let is_root_dir ops s0 d0_ref = (
  let root = (
    match ops.fops_get_root s0 with
    | Nothing -> (failwith "is_root_dir: impossible: no r
    | Just x -> x end)
  in
  ops.fops_dir_ref_eq s0 root d0_ref)
```

- Syntax OCaml-like; minor differences in syntax (`val`, `match...end`); explicit type quantification; additional support for writing theorem statements, executable tests etc.

Lem status

- The main developers have moved on, so not much activity
- The users seem to be happy enough
- The code is staggeringly complicated (eg Lem knows about how to compile pattern matches for each target back-end; so Lem knows about all the differences in pattern matching syntax between Isabelle/HOL and HOL4, and how to compile deeply nested patterns to shallower patterns etc)
- Some of the features seem to be a bit fragile (probably due to the complexity of what they tried to achieve)
- Thought: as with Ott, you could get a fair way towards Lem functionality with decent parsing/pretty-printing tools (although Lem is obviously much more sophisticated)
- We used Lem for SibylFS

Outline of talk

- Early work
- Netsem
- POPLmark
- Ott and Lem
- **SibylFS**
- Current/future work

What is SibylFS?

SibylFS is simultaneously

- A file system specification
- A test oracle that can be used to check real-world traces of file system behaviour

Existing specifications

- Everyone knows specifications are important; how else do you know what something is supposed to do? So, where are these specifications?
- For file systems:
  - POSIX online spec eg http://pubs.opengroup.org/onlinepubs/9699919799/
  - other informal specifications (Linux Standard Base, libc, muslc etc)
  - man pages (Linux, Mac, FreeBSD... all subtly different, even between distributions of same OS)
  - implementations (presumably you can read the source code and discover "the truth" for a particular impl)
  - test suites (for a small number of test cases eg 50 for rename in POSIX test suite)

Problems with existing specifications

- Problems:
  - lots of specifications - which is "right"?
  - specs are mostly informal, and so (almost inevitably) ambiguous, incomplete, plain wrong
  - kernel code is formal, but unreadable unless you are a kernel coder, and impractical as a spec
  - (perhaps most important) Does an implementation actually meet the specification? Unfortunately not possible to use existing specs to test an implementation directly

Problem: implementations and specifications

- Do file system implementations meet the specifications? **NO** (none of them as far as we can tell!)
  - for "typical" use cases, they mostly behave sensibly; in edge cases things go wrong
  - $=>$ the specifications are not much use in these edge cases

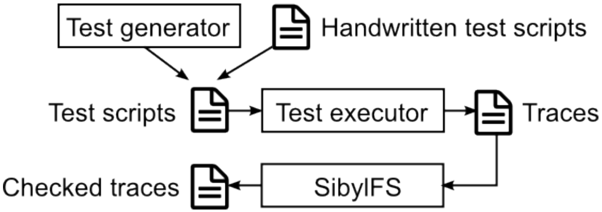
Another problem: how do implementations differ?

- In edge cases, we know file systems behave quite differently
- The question now is: how do they behave differently? Some individuals (e.g. file system developers) may know some of the truth (e.g. about their file system); no-one knows the full truth.

What have we done?

- A **formal specification** of filesystem behaviour (with variants for POSIX, Linux, Mac, FreeBSD)
  - precise, unambiguous, readable, maintainable, comphrehensive, detailed
- A **test oracle** (based directly on the spec) that can determine whether *any* observed trace of a real-world system conforms to the spec
- The test oracle enables exhaustive testing. We also provide an extensive test suite of $>$20k scripts, and the results of running those tests on $>$40 different combinations of libc+OS+filesystem
  - 20k test cases (and acceptable results) probably infeasible with hand-written test cases
  - impossible to compare results at this scale (800k traces) without an oracle

System structure



- *Test scripts* mostly generated automatically, with additional hand-written scripts
- Test scripts are fed to a *test executor* which executes the scripts on a real-world libc+OS+filesystem stack
- Results are recorded as *traces*
- Traces are checked by SibylFS (the spec) to determine whether they represent allowable behaviour or not; the output is in the form of a *checked trace*

Example test script

A test script is essentially a list of libc calls to execute on a real-world system

```
@type script
######################################
# Test link___link_nonempty_dir1__d2__sl_dotdot_d2___nonen
######################################
mkdir "empty_dir1" 0o777
mkdir "empty_dir2" 0o777
mkdir "nonempty_dir1" 0o777
mkdir "nonempty_dir1/d2" 0o777
open "nonempty_dir1/d2/f3.txt" [O_CREAT;O_WRONLY] 0o666
write! (FD 3) "Lorem ipsum dolor sit amet, consectetur adi
close (FD 3)
... // further setup commands
link "nonempty_dir1/d2/sl_dotdot_d2" "nonempty_dir1/nonex
...
```

Example trace (behaviour of real-world system)

A trace records the libc calls from the test script, and the responses received from the real-world system

```
# processing file 'link___link_nonempty_dir1__d2__sl_dotdo
        @type trace
        ######################################
        # Test link___link_nonempty_dir1__d2__sl_dotdot_d2_
        ######################################
     5: mkdir "empty_dir1" 0o777
        Tau
        RV_none
     6: mkdir "empty_dir2" 0o777
        Tau
        RV_none
     7: mkdir "nonempty_dir1" 0o777
        Tau
        RV_none
     8: mkdir "nonempty_dir1/d2" 0o777
        Tau
```

Example checked trace

- See e.g. here (local) or here (online)

The SibylFS specification

- A specification of the behaviour of libc+OS+filesystem, i.e., the behaviour a user process (application) sees when linked to libc
- Written in Lem; available in HOL4, Isabelle/HOL and OCaml; about 6k lines of Lem/HOL for the specification
- Variants for POSIX, Linux, Mac and FreeBSD
  - For POSIX, a spec of allowable behaviours; for Linux Mac and FreeBSD, a spec of existing real-world behaviours (with some looseness); the parts of the spec that are Linux/Mac/FreeBSD weird behaviours are clearly identifiable

Main differences to Netsem

- Effort to construct the spec: 3 person years vs 10+ years
- Spec clearly structured, modular, elegant (as far as possible); this makes it much easier to update the spec and debug it
- The oracle used to check traces is 6 orders of magnitude (!) faster than the Netsem oracle. Mainly this is due to the way the spec was written, which enabled direct extraction of a test oracle to OCaml

Labelled transition systems again

- For Netsem, we had a specification detailing transitions $s \xrightarrow{l} s'$
- Here s is the state of a network, including the state of the hosts.
- For file systems, it doesn't make sense to do this.

SibylFS trace checking

- We must specify the interface to file systems, not the internal state.
- We still use an LTS, but now the states are "abstract", with no direct relation to real-world file system states (and we don't want to specify what the relation is! there are too many real-world file system implementations).
- All that matters is the sequence of **labels** that the spec gives rise to

SibylFS trace checking

Given a sequence of observed events (labels)...

$$l_1 \; l_2 \; l_3 ...$$

...find a trace

$$s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ldots$$

Form of the transition system: a step function

Netsem specified transitions in the form

$$P(s, l, s') \longrightarrow (s \xrightarrow{l} s')$$

where P is not executable. For SibylFS, we specify a function *step* such that

$$step(s, l) = \{s' | P(s, l, s')\}$$

And this *step* function **is** computable.

Specification states are abstract, symbolic, with constraints

- They can be manipulated directly in OCaml

The oracle

- The oracle processes the labels one by one, keeping track of an infinite set of possible real-world states
- Given a sequence of observed events (labels) $l_1 l_2 l_3$... we compute $S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} S_2 \ldots$
- The S_i are sets of possible *spec* states. There are a finite number of spec states at each stage. Each spec state corresponds to possibly infinite real-world states.
- Pros: very fast trace checking
- Cons: spec arguably less natural; custom-built ad hoc symbolic expressions/ constraint languages (in Lem/HOL, and hence OCaml)

The main challenges when writing the spec

- Interpreting POSIX
- Writing the spec so that it can be used to efficiently check real-world traces of behaviour (the problem is nondeterminism and state space explosion)
- Dealing with the complexity of observed real-world behaviour

The spec

An html version is here

See e.g. `fsop_rename_checks_rsrc_rdst`

Testing

Testing

- The spec is reasonably large as a specification (c. 6000+ lines); how can we gain confidence that it is correct?
- From the beginning we wanted to extensively validate the spec, by using it to check traces of real-world behaviour
- This form of testing also uncovers bugs in real-world systems

Test oracle enables combinatorial testing

- Existing filesystem test suites hardcode the expected answers for a given libc call; practically, they tend to have a relatively small number of tests
- Our approach is different: a test script is just a sequence of libc calls (we don't need to say what should happen after each call - the spec already contains this knowledge)
- This enables randomized and combinatorial testing
- We try to exhaustively combinatorially test the libc interface using tests that are generated automatically
- Much more usable and much less effort than hand-coding test cases (hand-coding is infeasible at this scale)

Difficulty of LTS trace checking

- NetSem gave a specification of UDP and TCP/IP as an LTS which was then used as a test oracle; our approach is broadly based on the NetSem approach
- NetSem took 2500 CPU-hours to check 1000 traces; this is at the limit of practicality; the cost of checking made it very difficult to update and revise the spec
- Checking a trace against an LTS is a very general problem which I think deserves a bit more attention
- The SibylFS spec was designed from the start for efficient trace checking: checking >20k tests on a 4-core i7 takes about 79s (it takes 152s to execute the tests on an in-memory tmpfs filesystem)
- Our testing involves a very large number of test scripts; checking is extremely fast; indeed SibylFS is fast enough that it could be used to check behaviour "online"

Test results

We found the following sorts of bugs

- Errors (and ambiguities etc) in the specifications (including POSIX, man pages etc)
- Errors and deviations in implementations
- Errors in our spec (which we then fixed of course!) and tracing infrastructure

Test results, stats

- Trace acceptance
  - Linux: 21061 accepted by spec; 9 rejected (21070 total)
  - Mac: 21036 accepted by spec, 34 rejected (21070 total); FreeBSD similar
  - essentially no barrier to getting 100% trace acceptance
- Coverage: $>$98% (of the spec)
  - by way of comparison, a paper (Groce et al., 2007, "Randomized differential testing…") from NASA scientists that applied randomized testing to a model of a filesystem achieved 89% coverage

Test results, strange behaviours

- Error codes are quite often non-POSIX
- Path resolution, particularly when a trailing slash is involved, is variable, non-POSIX
- Treatment of paths referencing symlinks, particularly when the path ends in a trailing slash, is highly variable
- Various overlay filesystems, and FUSE filesystems, mess up things like permissions

Test results, strange behaviours

- More serious: posixovl/VFAT (posix emulation on top of VFAT) gets link count wrong when rename overwrites a file that is linked elsewhere; possible to get to a state where the filesystem contains no files, but there is no free space (space leak)
- OpenZFS on Linux 3.13.0-34 (Ubuntu Trusty): files opened with `O_APPEND` would not seek to the end of the file before `write` or `pwrite` (probably causing applications that use this functionality to fail)
- OpenZFS on OS/X: possible to execute a sequence of calls which leads to the calling process hanging using 100% CPU, unresponsive to signals; volume cannot be unmounted, machine cannot be shut down; force unmounting may cause storage device to become unusable until next system restart
- Permissions: implementations of permissions should give the same behaviour from one kernel version to the next; however, we found tests involving file and directory access that failed on Linux kernel 3.13 and succeeded on 3.14; pronbably a buggy 3.13 implementation of permissions in edge cases

A FreeBSD bug

- An important invariant
  - if there is an error when executing a file system function, then the state of the file system is unchanged
  - => e.g. so if I try to create a file using open and I get an error, I don't have to clear up after myself
  - this invariant appears to hold for POSIX, Linux and Mac
- https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=202892 (2015-09-04!)
  - opening a file: symlink deleted, file created, error returned
  - => here, this invariant is broken on FreeBSD

An FSCQ bug

https://github.com/mit-pdos/fscq-impl/issues/2

dsheets commented on 3 Oct 2015

On an empty file system built from `27c4078`, running

```
#include <unistd.h>
#include <fcntl.h>

int main() {
  close(creat("spaaaaaace"));
  truncate("spaaaaaace",10);

  return 0;
}
```

results in a file, `spaaaaaace`, which contains the text `spaaaaaace` instead of 10 null bytes. For applications which expect POSIX, OS X, Linux, or BSD behavior when using `truncate` to extend files, this unexpected data can result in incorrectly formatted files.

zeldovich commented on 3 Oct 2015 Owner

Thanks; I will take a look. Sounds like we might have gotten the spec wrong.

Testing summary

- Huge number of tests, mostly automatically generated; tested on a large number of stacks
- SibylFS is very efficient at checking tests
- Excellent trace acceptance and coverage figures
- In edge cases there are numerous differences between POSIX, Linux, Mac and FreeBSD; most are not very interesting, but the spec gives a complete description of them; the testing even uncovered some relatively serious bugs (which we didn't expect) and (please don't repeat this) a serious bug in a verified file system
- The real result of testing is that we have confidence in the spec

A virtuous circle

- Is our spec/ testing perfect?
- Almost certainly not, but in many ways it improves on the status quo
- And there is a virtuous circle

Thoughts on specification and validation of large real-world systems

- I feel I am almost at the point where I know how to do this (SibylFS is not quite right)
- The problems are mainly software engineering (modularity, separation of concerns), and ensuring the spec is efficiently checkable (which mainly requires deciding how to handle all the forms of nondet)
- Clean slate spec and validation could be much simpler (e.g., one could determinise many cases to reduce nondeterminism and make checking easier)

What next?

My work is mainly on:

- Verified file system implementation (based on novel proofs about B-trees)
- Parsing

Verified file system

We are currently working on **a verified file system implementation**. Compared to existing file systems, we hope for:

- better performance
- more features and functionality
- verified correctness

We note that

- B-trees are a key datastructure; existing proofs are not suitable for formalization (especially with concurrency); we have developed new proofs of correctness and are in the process of formalizing these

Parsing

- I want a parsing tool that can replace latex, Ott and Lem
- That is very simple, but fast
- That can parse "infinite" grammars, and mildly-context-sensitive grammars e.g. markdown
- That has a re-targettable front-end (so you can use any formalism you like to specify your parsers - cf. grammar zoo, IETF BNF spec RFC7405 etc)
- Currently I am thinking about associativity and precedence and ways to combine them with combinator/Earley parsing

Overall aim

- Better theorem provers and related tools
- Portable definitions and proofs so we don't have to keep reinventing the wheel; portable verified implementations
- Simplicity, beauty and elegance

Current working style

- I have started to use Scala because of the libraries and tools; I tend to prototype and test code in Scala, and later port to Isabelle
- I try to capture as much proof structure in Scala as possible e.g. not only the code, but any abstractions, invariants etc
- For testing, I try to test all invariants etc that I can; this is extremely useful for catching bugs.
- I am thinking of writing a lightweight proving tool in Scala which uses off-the-shelf rewriting tools and first-order-provers (with my parsing technology as the front-end)
- "Modularity matters most"; "separation of concerns"

THE END. Thank you for listening