SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems

Dr Tom Ridge

2016-01-27

Outline of talk

Part I

- Historical background

Part II

- Netsem TCP/IP specification
- SibylFS file system specification
- Throughout: what things did we learn?

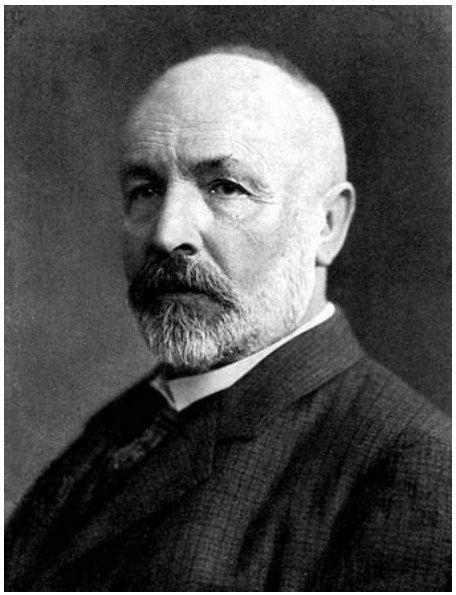
Logic

What is a proof?

Euclid (c. 350 BC - 300 BC)

Cantor (1845 - 1918)

"No one shall expel us from the Paradise that Cantor has created." (Hilbert)



Spur for logic

Infinite sets, unclear

Logic really gets going

Some things are completely clear: finite objects (trees, graphs, symbols)

Proofs as formal objects (trees)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[y : A \wedge B]}\,(\mathrm{A})}{A}\,(\wedge\mathrm{E}) \quad \dfrac{}{[x : A \Rightarrow B \Rightarrow C]}\,(\mathrm{A})}{B \Rightarrow C}\,(\Rightarrow\mathrm{E}) \quad \dfrac{\dfrac{}{[y : A \wedge B]}\,(\mathrm{A})}{B}\,(\wedge\mathrm{E})}{C}\,(\Rightarrow\mathrm{E})}{A \wedge B \Rightarrow C}\,(\Rightarrow\mathrm{I,y})}{(A \Rightarrow B \Rightarrow C) \Rightarrow (A \wedge B \Rightarrow C)}\,(\Rightarrow\mathrm{I,x})$$

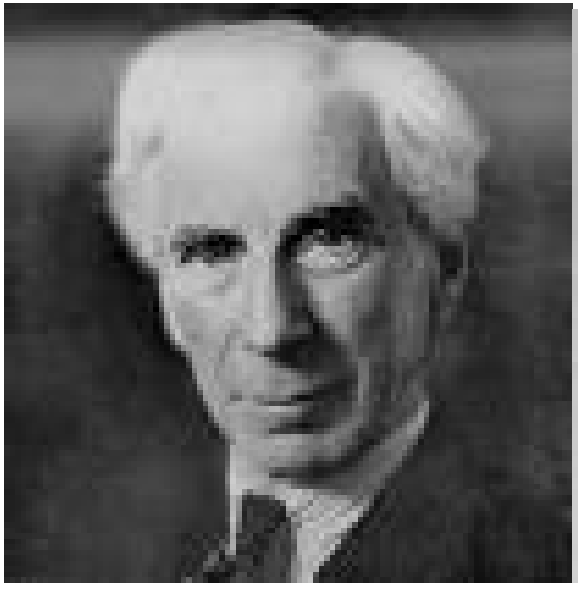
First-order logic

$\forall x.P(x)$ and $\exists x.P(x)$

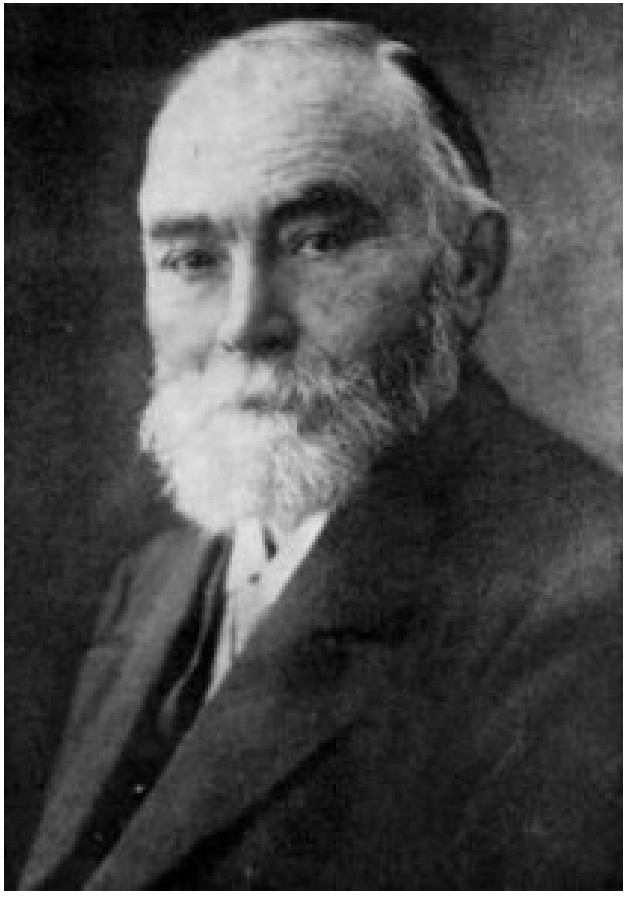
Set theory

First-order logic is not enough for **mathematics**

Bertrand Russell




Gottlob Frege



Your discovery of the contradiction caused me the greatest surprise and, I would almost say, consternation, since it has shaken the basis on which I intended to build arithmetic. (Frege to Russell)

Russell on Frege's acknowledgement

Penrhyndeudraeth, 23 November 1962

Dear Professor van Heijenoort,

I should be most pleased if you would publish the correspondence between Frege and myself, and I am grateful to you for suggesting this. As I think about acts of integrity and grace, I realise that there is nothing in my knowledge to compare with Frege's dedication to truth. His entire life's work was on the verge of completion, much of his work had been ignored to the benefit of men infinitely less capable, his second volume was about to be published, and upon finding that his fundamental assumption was in error, he responded with intellectual pleasure clearly submerging any feelings of personal disappointment. It was almost superhuman and a telling indication of that of which men are capable if their dedication is to creative work and knowledge instead of cruder efforts to dominate and be known.

Yours sincerely,

Bertrand Russell

The translation of Frege's letter is by Beverly Woodward, and it is printed here with the kind permission of Verlag Felix Meiner and the Institut für mathematische Logik und Grundlagenforschung in Münster, who are preparing an edition of Frege's scientific correspondence and hitherto unpublished writings; this edition will include the German text of the letter.

Church

Recursive functions, lambda calculus

Turing

Turing machine

The computer

Computers, logic and proof

Proof search

Any mathematical proof that can be proved in set theory can be proved by computer (just enumerate proofs)

Is the human being irrelevant?

Is the human being irrelevant? Godel: maybe not

Naive: "For a given formal system, there will always be truths that are not provable"

Is the human being irrelevant? Practically: no

Computer proof search is too slow for non-trivial properties

Are humans just optimized forms of proof search which could be carried out on computer?

No, the human is clearly being creative, which is quite different from proof search on a machine

Human "mathematical" intelligence may eventually be replaced by machines, but this is not a certainty

Especially at the highest levels of mathematics
And especially when it comes to modelling situations mathematically, or giving specifications, or stating lemmas

Russell's experience: don't do this by hand

The siren song of certainty

When the rules of logic and proof are coded on the computer we can be reasonably confident (as confident as we can be) that the proofs are formally correct

This is really what grabbed me and made me want to do research in this area

If you want programs that are correct, prove them correct... using a computer

Today (say, 2005 onwards): powerful, cheap computers

Consequently theorem provers start to be applied in many areas

Theorem provers still painful to use

- User interfaces etc (Emacs released in 1976 !)

UI progress?

25 years ago I hoped we would extend Emacs to do WYSIWG [sic] word processing. That is why we added text properties and variable width fonts. However, more features are still needed to achieve this.
Could people please start working on the features that are needed? (Richard Stallman, 2013)

Theorem provers still painful to use

- IMHO: the HOL logics are practically not very suitable for large-scale formalization and probably not viable in the long run (what we need is something like Set theory as the foundation, and various typed systems on top, with facilities for proving "in the large" - modules, traits etc)
- Libraries are poor; effectively no reuse of proofs between systems (but how much reuse is there of libraries written in different programming languages?)
- But, for "economic" reasons, it is almost inevitable that theorem provers will become standard tools in many areas of mathematics (just as open source has become widely used)

What should we prove correct?

Huge effort, so focus on some key components
e.g. operating systems, networking, file systems, compilers, and tools such as parsers

The problem with verified software implementations

- Often slow (target language usually has GC; if not, verification effort huge)
- Not widely used (because the target language is not widely used)
- Still mostly prototypes
- Compcert verified compiler for a C-like language
  - a good target: we do not really care that the compiler is a bit slow, or is written in, say, OCaml
  - but not widely used
- But we are almost there for software; for hardware, of course, Intel famously already verifies quite a lot of the chip design

Another problem: specifications

- Everyone: “specifications are really important!”
- But where are these specifications?
- What does it mean, say, to prove that TCP/IP networking is correct?

How can we provide better specifications?

How can we validate real-world systems against these specifications?

These are fundamental and important questions

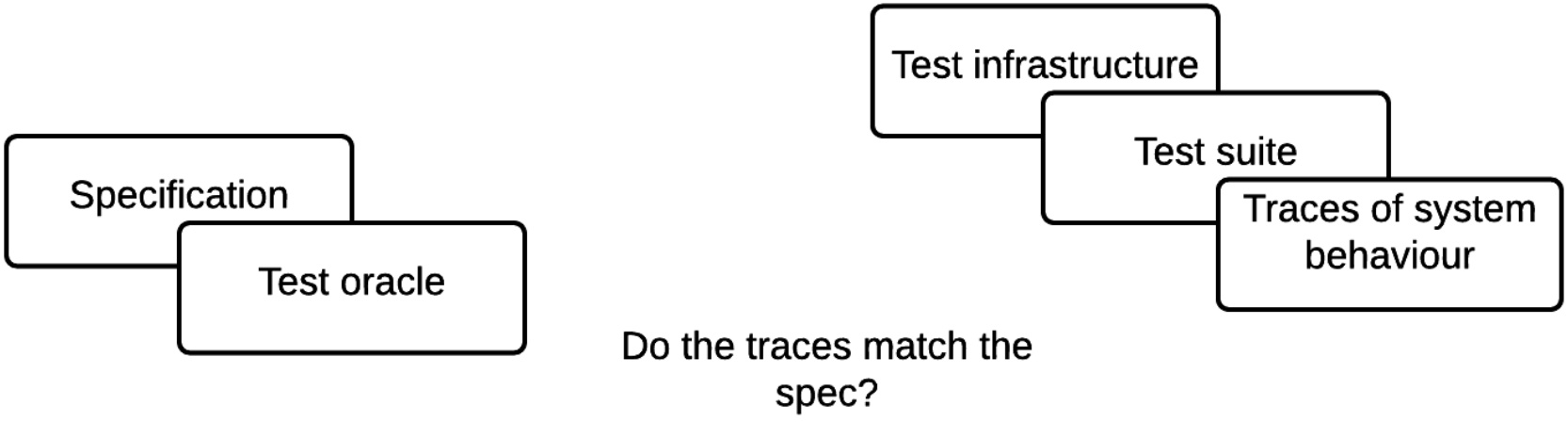
Timeline

- Netsem: c. 2001 - 2009
- SibylFS: c. 2013 - 2015

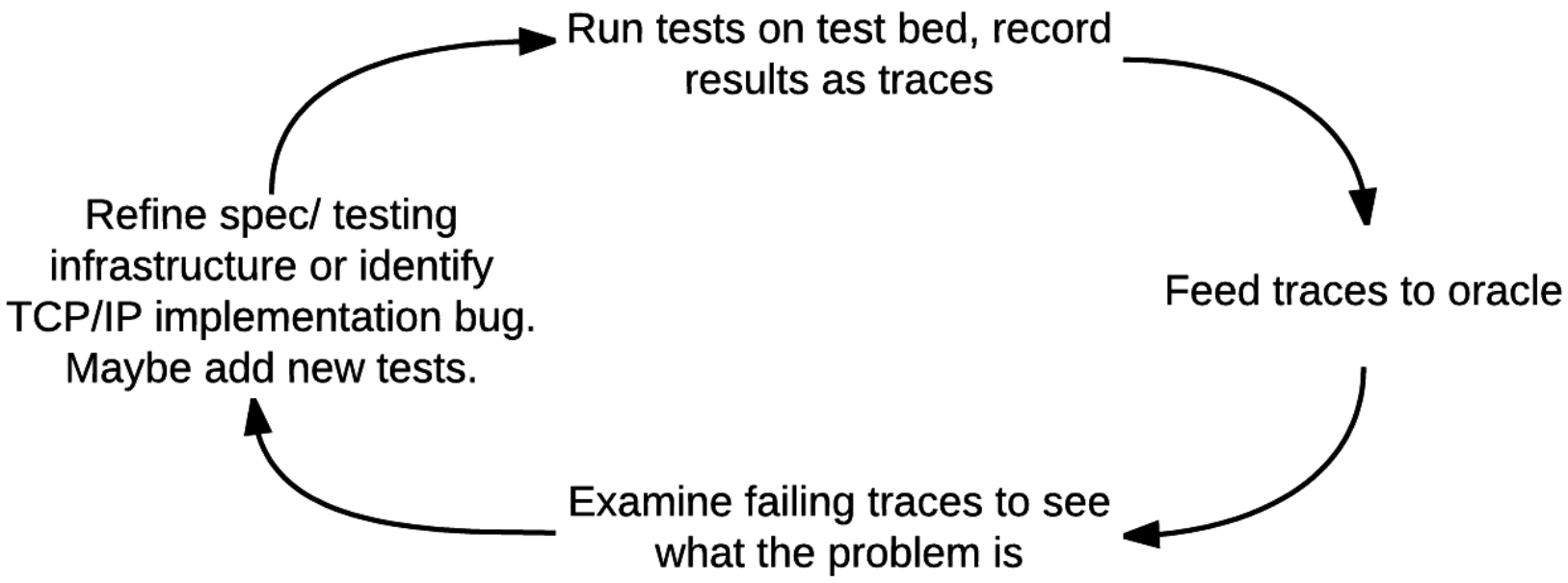
Netsem TCP/IP specification

- Concurrency, distribution clearly important
- 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
- 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**

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

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

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 and record traces. Logging of events on wire, in TCP/IP stack and at sockets API
- 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: $P(s, l, s') \rightarrow (s \xrightarrow{l} s')$
- 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

Example trace

https://github.com/PeterSewell/netsem/blob/master/Net/TCP/Demo-traces/trace1741#L87-L112
Or here

Problems

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:
$s_0 \xrightarrow{l_1} s_1(\mathit{partial}) \xrightarrow{?} s_2 \ (\xrightarrow{?})?\ s_? \ (\xrightarrow{?})?\ s_i \ldots$
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 symbolic execution engine of HOL, and the HOL specification

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

Possible solution

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.

Problems

- Suppose we have $s \xrightarrow{I} s'$ where we have full information about s, I, s'.
- The specification is given in the form $P(s, I, s') \rightarrow (s \xrightarrow{I} s')$
- This is a **specification**. There is no guarantee that it is possible to compute whether $P(s, I, s')$ 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

Several years later. . .

The specification tar pit

- I wanted to verify a novel file system implementation
- So I needed a file system specification
- So I started to construct a "little" specification, but to get somewhere plausible takes significant effort
- Especially if you are committed to validating the spec
- And as soon as you start valdiating, you find problems, which means you have to fix the spec
- And eventually you are forced to produce a reasonably extensive, validated spec

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.

Do specifications really matter? Do file systems matter?

See here "Speaking with my 'git' hat on. . . "
And more recently http://www.itworld.com/article/2868393/linus-torvalds-apples-hfs-is-probably-the-worst-file-system-ever.html

Netsem redux

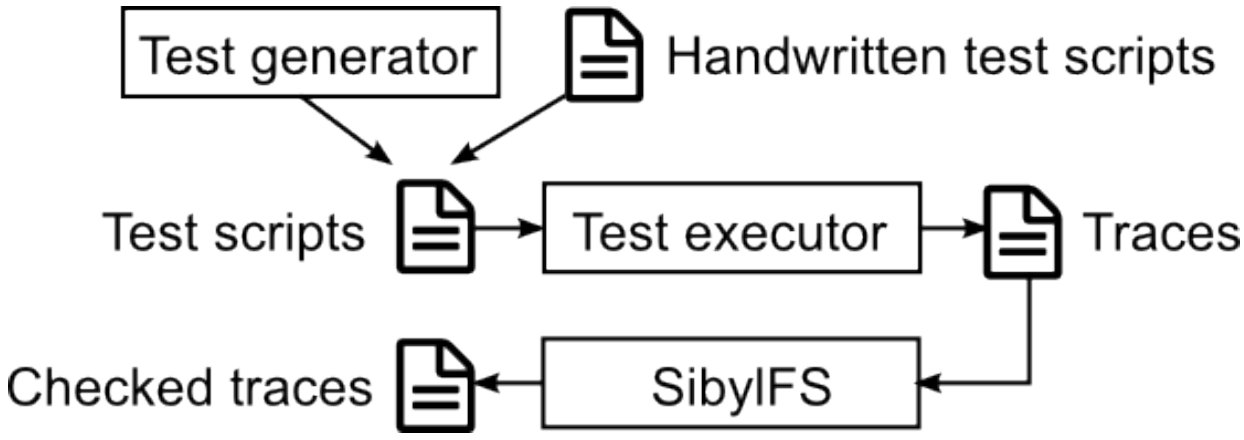
What do we want from a specification?

- Precise, unambiguous, readable, maintainable, comphrehensive, detailed
  - also loose: the spec should precisely and unambiguously describe any looseness; but not too loose...
- Usable as a test oracle
  - ideally it should be easy to use the spec directly to test conformance of real-world implementations

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___nonem
#########################################
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/nonexi
...
```

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

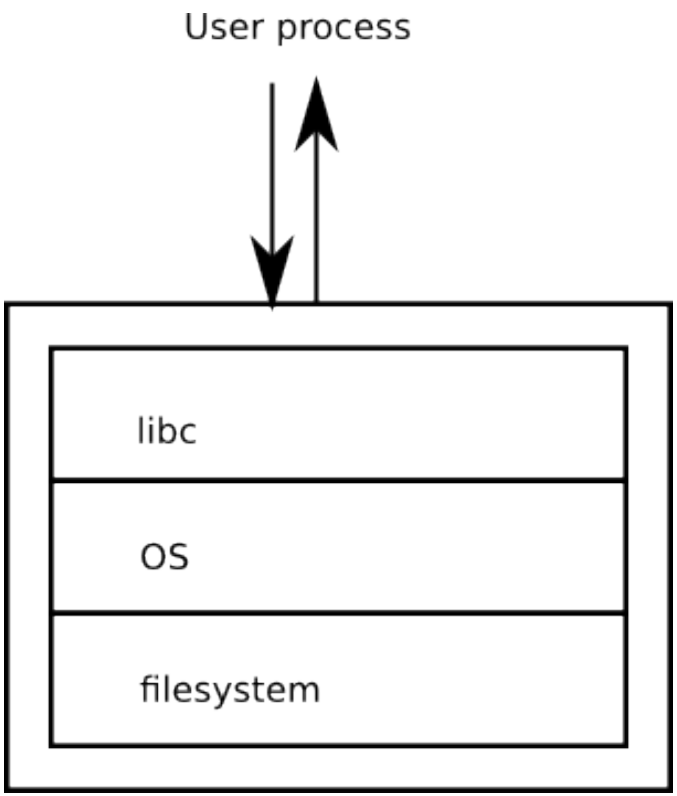
- A specification of the behaviour of libc+OS+filesystem, i.e., the behaviour a user process (application) sees when linked to libc
- Written in Lem; available in HOL4, Isabelle/HOL and OCaml; about 6k lines of Lem/HOL for the specification
- Variants for POSIX, Linux, Mac and FreeBSD
  - For POSIX, 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.

Form of the specification: labelled transition system

- What are the labels? (Labels correspond to the events we are interested in)
  - Interactions between a user process and the libc+OS+filesystem stack at the libc interface (`rename p1 p2` etc)
  - Process creation and destruction
  - Tau events (e.g. internal OS processing)



Top-level type of labels

```
type os_label =
  | OS_CALL of (ty_pid * ty_os_command)
  | OS_RETURN of (ty_pid * error_or_value ret_value)
  | OS_CREATE of (ty_pid * uid * gid)
  | OS_DESTROY of ty_pid
  | OS_TAU
```

- We specify the behaviour of the stack under any sequence of labels; multiple processes can execute concurrently, calls can overlap in time

libc calls

- What are the libc calls we handle? No memory-mapped files, no *-at forms of libc calls, no "filesystem out of space" errors, but mostly everything else related to files:

```
type ty_os_command =
  | OS_CLOSE of ty_fd
  | OS_LINK of (cstring * cstring)
  | OS_MKDIR of (cstring * file_perm)
  | OS_OPEN of (cstring * int_open_flags * maybe file_per
  | OS_PREAD of (ty_fd * size_t * off_t)
  | OS_READ of (ty_fd * size_t)
  | OS_READDIR of ty_dh
  | OS_OPENDIR of cstring
  | OS_REWINDDIR of ty_dh
  | OS_CLOSEDIR of ty_dh
  | OS_READLINK of cstring
  | OS_RENAME of (cstring * cstring)
  | OS_RMDIR of cstring
  | OS_STAT of cstring
```

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

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

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.

States are abstract, symbolic, constraints; directly executable in OCaml

The oracle

- The oracle processes the labels one by one, keeping track of a set of possible states
- Given a sequence of observed events (labels) $l_1 l_2 l_3 \ldots$ we compute $S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} S_2 \ldots$
- These 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

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

Constructing the model from real-world traces

Given an observed sequence

$$2, 3, 5, 7, 11, 13, 17...$$

Come up with the defn

```
let primes = ...
```

But with 800000 observed sequences, and an extremely complicated state space

The complexity of real-world behaviour

- A scenario: 18 months into the project, the POSIX and Linux variants of the spec were mostly complete; we wanted to extend the spec to cover Mac
- We ran the existing test scripts on Mac and checked them using the initial version of the Mac spec, derived from POSIX
- >20k test scripts; of the resulting traces, thousands failed to check (>5000 for `open` alone)
- These failing traces have to be examined and the spec updated so that the spec behaviour matches the real-world behaviour
- We need to:
  - figure out why the traces are failing
  - figure out what changes are needed to the spec; the changes should preserve the structure, readability and concision of the spec; not be too loose
- What helped: speed of trace checking; modularity: we tried to make the spec as modular as possible, with each component having a clearly defined role, with clearly defined interfaces etc.

The spec

An html version is here

See e.g. `fsop_rename_checks_rsrc_rdst`

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 less effort than hand-coding test cases

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: the Linux implementation of permissions should give the same behaviour from one kernel version to the next; however, we found test scripts involving file and directory access that failed on Linux kernel 3.13 and succeeded on 3.14 which we believe is due to a buggy 3.13 implementation of

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

Possible uses of the spec

- Documentation (formal counterpart to POSIX)
- Testing existing/new filesystems (particularly non-traditional filesystems) e.g. FSCQ, Flashix
- Reference specification for verified filesystem implementations
- Reference model (e.g. for systems research - a lot of papers construct their own little models of parts of the filesystem)
- Basis for formal proof (Thomas Tuerk did some initial work on this in Isabelle/HOL) e.g. of properties of filesystems, or as a basis for soundness proofs of program logics etc.
- To provide a POSIX compatiblity layer e.g. on Windows, or non-traditional settings such as Mirage
- For analysis of applications: e.g. do any applications use libc calls in edge cases where the behaviour of Linux and Mac differs?
- For analysis of filesystems: e.g. do Linux and Mac behave the same when restricted to some subset of libc?
- To identify gaps in existing test suites
- etc

Future work

Lots of possibilities. In the short term, we hope for take-up from industry.
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