Engineering with Logic: HOL Specification and Symbolic-Evaluation Testing for TCP Implementations

Steve Bishop* Matthew Fairbairn* Michael Norrish†
Peter Sewell* Michael Smith* Keith Wansbrough*
*University of Cambridge Computer Laboratory †NICTA, Canberra
http://www.cl.cam.ac.uk/users/pes20/Netsem

Abstract

The TCP/IP protocols and Sockets API underlie much of modern computation, but their semantics have historically been very complex and ill-defined. The real standard is the de facto one of the common implementations, including, for example, the 15 000–20 000 lines of C in the BSD implementation. Dealing rigorously with the behaviour of such bodies of code is challenging.

We have recently developed a post-hoc specification of TCP, UDP, and Sockets that is rigorous, detailed, readable, has broad coverage, and is remarkably accurate. In this paper we describe the novel techniques that were required.

Working within a general-purpose proof assistant (HOL), we developed *language idioms* (within higher-order logic) in which to write the specification: operational semantics with nondeterminism, time, system calls, monadic relational programming, etc. We followed an *experimental semantics* approach, validating the specification against several thousand traces captured from three implementations (FreeBSD, Linux, and WinXP). Many differences between these were identified, and a number of bugs. Validation was done using a special-purpose *symbolic model checker* programmed above HOL.

We suggest that similar logic engineering techniques could be applied to future critical software infrastructure at design time, leading to cleaner designs and (via specification-based testing using a similar checker) more predictable implementations.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; C.2.2 [*Computer-Communications Networks*]: Network Protocols; C.2.6 [*Computer-Communications Networks*]: Internetworking—Standards (e.g., TCP/IP)

General Terms Documentation, Design, Standardization, Theory, Verification.

Keywords Network Protocols, TCP/IP, Sockets, API, Specification, Conformance Testing, Higher-order Logic, HOL, Operational Semantics

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.

1. Introduction

Background The TCP/IP network protocols are among the most widely-used software infrastructure, with protocol endpoint implementations running on almost all machines. By and large the deployed Internet —the assembly of all these endpoints and the interconnecting routers— works remarkably well. When one looks at what the protocols *are*, however, and at what the behaviour of that endpoint code is (or should be), the situation is very unclear.

There are specifications: RFCs that focus on the on-the-wire protocols, and the POSIX standard for the Sockets API used by applications. These are prose documents: they describe the formats of wire messages and the C-language types of API calls precisely, but they are (almost inevitably) ambiguous and incomplete descriptions of the behaviour.

On the other hand, there is the code. There are many implementations of TCP/IP and the Sockets API. For each, the code does (implicitly) define some precise behaviour, but there are many differences between them, some intended and some not. The common implementations together form a de facto standard: any implementation must interoperate reasonably well with all of them, though the BSD implementations have a special status, with various protocol features first developed there. Moreover, each implementation is in itself a complex body of code. They are typically written in C, intertwined with an operating system. They are multithreaded, dealing with asynchronous events on the network interface, concurrent Sockets API calls, and the expiry of various timers. There is a rough layer structure (Sockets/TCP/IP/interface) but much coupling between the layers, with 'fast path' optimisations for the common cases, indirection via function pointers, and many historical artifacts. The deployed base makes it almost impossible to change the implementation behaviour (at either wire or Sockets interface) in any substantial way.

Developers, both of protocol stacks and of applications above them, thus have to deal with a very complex world. TCP is recognised as hard to implement correctly [19], [RFC2525], and indeed there is no precise sense in which an implementation is 'correct' or not. Application writers using the Sockets API have to be aware of a host of behavioural subtleties and implementation differences, in addition to the intrinsic difficulties of concurrency, partial failure, and malicious attack. It it clearly possible to write reasonable protocol stacks, and distributed libraries and applications above them, but the cost and level of expertise needed are high.

Problem We set out to answer two questions: the specific question of what this de facto standard is; and the general question of what specification and validation techniques are needed to treat the behaviour of such systems rigorously.

The first should provide a basic reference for developers of protocol stacks and of distributed applications above the Sockets API, and a basis for formal reasoning about either. The second aims towards making it feasible to precisely specify the behaviour of future protocols (or similarly critical software infrastructure) at design time, and also to enable automated conformance testing from those specifications, and hence ultimately lead to a more comprehensible and robust software environment than we have now.

Contribution In answer to the first question, we have produced a 'post-hoc' specification of TCP, UDP, and the Sockets API that reflects the behaviour of several existing implementations (FreeBSD 4.6, Linux 2.4.20-8, and Windows XP SP1). It is fully *rigorous*, expressed in higher-order logic. It is *detailed*, with almost all important aspects of the real-world communications at the level of individual TCP segments and UDP datagrams, with timing, and with congestion control (it abstracts from the internals of IP). It has broad *coverage*, dealing with the behaviour of a host for arbitrary incoming messages and Sockets API call sequences, not just some well-behaved usage — one of our main goals was to characterise the failure semantics under network loss, duplication and reordering, API errors, and malicious attack. It is also remarkably *accurate*, experimentally validated against those implementations (though TCP validation was mainly with respect to BSD). The specification is available as a technical report, together with an overview of the project [6, 7].

In this paper we focus mainly on the second question, discussing the specification language and idioms we needed and describing the validation technology we developed. The fact that our techniques suffice for the post-hoc specification work we have carried out is a strong indication that they could be used at design time for future systems.

A companion paper gives a systems-oriented view of the work, focussing on protocol and API modelling issues, the level of abstraction used, anomalies found in the implementations, and lessons for future protocol design [5]. We build on our earlier work [20, 21, 26, 18] which involved only much simpler UDP models, not covering TCP and without HOL-based checking.

Approach The key to our approach is the use of an expressive logic supported by a general-purpose mechanized proof assistant, the HOL tool [11, 12]. HOL supports classical higher-order logic. It provides a rich type structure, including numeric types and user-definable datatypes, with ML-style polymorphism and type inference. The system provides the programmer with a variety of decision procedures and scriptable tactics. HOL is not a fully automatic theorem-prover or model checker, as higher-order logic is not decidable, but its programmability allows the development of stand-alone tools, tailored to particular domains.

The specification is written as an operational-semantics definition in higher-order logic, with various idioms adapted from programming language and concurrency theory semantics, including a relational monad structure and nondeterministic timed labelled transition systems. An important goal in writing the specification was to make it as readable as possible (in sharp contrast to the code, which has evolved over the years and has considerable performance optimisation). We discuss these idioms in §2.

The specification was produced and validated with an *experimental semantics* approach. We produced an initial draft specification based on the RFCs [RFC768, 791, 792, 793, 1122, 1323, 2414, 2581, 2582, 2988, 3522, 3782], POSIX [13], standard texts [24, 28, 25], BSD and Linux source code, and ad-hoc tests. In parallel, we instrumented a test network (containing machines running each of the three implementations mentioned above) and wrote tests to drive those implementations, generating several thousand real-world traces chosen to cover as much of their behaviour as we could. We then ensured that the specification admitted those traces by running a special-purpose symbolic model checker, correcting the specification when we discovered that particular real-world behaviour was not included. This was a computationally-intensive activity, and so checking had to be distributed over many machines.

Our symbolic model checker takes a captured trace and checks whether it is included in the set of all traces of the specification. Rather different from conventional model-checking (symbolic or otherwise), the states here are arbitrary higher-order logic formulae, essentially constraints on the underlying state of a protocol endpoint. As the checker works along a trace (possibly backtracking) it uses various HOL tactics, e.g. for simplification, to symbolically evaluate the specification. Lazy control of the search through the tree of possibilities is essential. The checker either succeeds, in which case it has essentially proved a machine-checked theorem that that trace is included, or fails, for a trace that is not included, or terminates if one of several heuristic conditions is satisfied. HOL is a proof assistant in the LCF style [10], and so its soundness, and the soundness of the checker above it, depends only on the correctness of a small trusted kernel. The checker is described more fully in §3.

The results of trace checking are summarised in §4, which briefly recapitulates some of the experimental procedure and data from [5], and we discuss further related work and conclude in §5 and §6.

Contrasts Our approach is rather different to most previous work on program verification and model-checking. We combine the use of a general-purpose mechanized proof assistant with what can best be described as logic engineering, both in the specification and in the checker. We have developed techniques, some principled and others ad-hoc, that together pragmatically suffice for this problem; we leave for future work the task of understanding how far they can be generalised. In the remainder of this section we highlight the key aspects of the problem that have led us to these choices.

Post-hoc vs pre-hoc specification Traditionally one thinks of testing an implementation against a pre-existing specification. Here, faced with the entrenched de facto standard of the deployed implementations, the best that can be done is identify what their behaviour is — hence our post-hoc experimental semantics approach. The checker technology, however, is symmetric: it could equally well be used to test future implementations against our now-existing specification. Indeed, our trace-checking threw up a number of behaviours in the implementations that should almost certainly be classed as bugs, and many differences between the three implementations. Some of these are described in §4.

Nondeterminism The variation in extant implementations, the liberality of existing specifications permitting much of this variation, and the fact that the behaviour of any one implementation is highly dependent on OS scheduling, timers, and so on, mean that it is not enough to specify just one "correct" behaviour. Instead, our formal specification must reflect the informal specifications' looseness and describe a wide range of legitimate behaviours.

This need for nondeterminism strongly influences the choice of language in which to write the specification and the checking techniques that can be used. Our specification is quite different from a "reference implementation" for TCP in a more-or-less conventional programming language, which would describe just one behaviour of the many possible (several of these exist, including the BSD C code and those by Biagioni in Standard ML [4], by Castelluccia *et al.* in Esterel [8], and by Kohler *et al.* in Prolac [15]). Instead,

we have had to express constrained behaviour in a relational style, within the higher-order logic of HOL. Moreover, much nondeterminism is *internal*, observable only indirectly after several further interactions. Checking conformance cannot be done by simply running the specification and an implementation in lock-step and comparing the results, but needs the more sophisticated methods of §3.

Code scale and complexity Ideally one would want to prove that all executions of the implementation code meet the specification, rather than testing that a sample of traces do. Unfortunately, the scale and complexity of the code seem to be prohibitive. For example, the relevant part of the BSD code is of the order of 15 000 – 20 000 lines of C. To the best of our knowledge, there is not even a formal semantics for the fragment of C that is used, let alone proof techniques that have been shown to scale to this kind of problem. Moreover, the relevant code is intertwined with the remainder of the OS.

Specification scale Another tempting goal would be to prove correctness results for the protocol (as made precise by the specification). For example, one might prove that two copies of the endpoint-specification, in parallel with a simple model of the network and under assumptions about the amount of message loss etc., really did provide a reliable-stream communication service. Even more ambitiously, one might like to prove that the TCP congestion control algorithms (as described in the specification) are effective, leading to (reasonably) stable and fair global network behaviour.

Both of these again founder on the problem of scale. The specification is a moderately large document (386pp typeset; 25 800 lines of HOL source, of which around $^2/_3$ are comment). Proof about such a large definition is a daunting prospect, but our pragmatic specification-based testing approach scales reasonably with the size of the definition. It is also independent of the size of the code, though not of its behavioural complexity.

Rich properties Other model-checking techniques have been applied to substantial bodies of real code, notably BDD-based methods (in the more usual sense of 'symbolic model-checking') and predicate abstraction methods. They are generally focussed on detecting runtime errors, such as dereferencing null pointers and assertion violations. In contrast, we check conformance with a complete specification of the system behaviour, a much more elaborate property which would be hard to state with assertions or simple temporal logics. On the other hand, those methods analyse the source code directly, whereas we consider only its behaviour as manifested in the generated traces.

State space A TCP endpoint has a very large and complex state space, in the implementations and in the specification. In the latter, the control block for each end of a TCP connection contains 14 32-bit sequence numbers, as many natural numbers again, and 10 timers, represented with real numbers. An implementation would likely use 32-bit numbers instead of the specification's unbounded natural numbers, and types at least this wide for the timers. A rough estimate thus suggests that each connection would take 1200 bits to model if a finite translation were attempted.

Further, the specification allows for an arbitrary number of connections to be made, and for arbitrary number of messages to be in various of the host's queues. Ignoring the data being transmitted in the packets and their IP addresses, each TCP segment contributes another 190 bits to the size of the state.

Clearly, any finite analysis of the specification would have to dramatically constrain its possible behaviours, and would also require a sound translation from the high-level specification to the finite model. Both of these requirements are unpalatable.

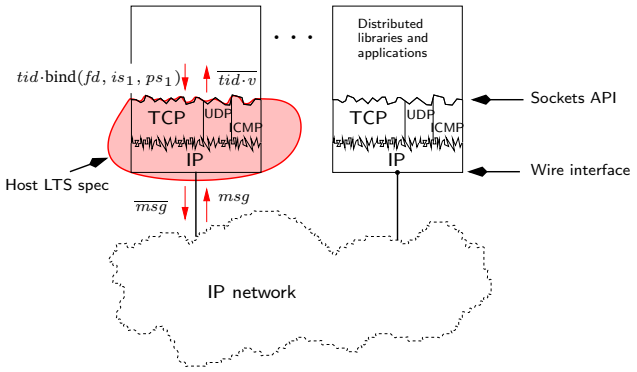
No essence Finally, it is worth emphasising that we are working with TCP as it actually is, that we approach it as an experimental object. We are not trying to distill some simple 'Platonic essence' of TCP. Indeed, it is not clear that it has one in any useful sense. The protocol has many aspects: connection setup and teardown (as loosely described in the classic 'TCP state diagram' [27]), sliding-window flow control, congestion control, protection against wrapped sequence numbers, round-trip time estimation, protection against certain denial-of-service attacks, and so forth. These are intertwined in subtle ways, with almost no modular structure.

Programmers writing TCP/IP stacks and systems on top of TCP need to understand it at an intuitive level, but crucially also need to understand the warts and wrinkles of its actual implementations. Not all aspects are important in all circumstances, but all are important in some.

2. Specification Structure and Idioms

In this section we describe the form of the specification and the main idioms (within higher-order logic) that have made it feasible to write it. By working in a general-purpose proof assistant we have been able to choose specification idioms almost entirely on clarity, not on their algorithmic properties. The following section addresses the algorithmic and proof aspects of checking traces against this specification.

External form The main part of the specification (modelling the shaded region below) is the *host labelled transition system*, or *host LTS*, describing the possible interactions of a single host OS: between program threads and host via calls and returns of the Sockets API, and between host and network via message sends and receives.



The host LTS is a transition relation $h \xrightarrow{l} h'$, where h and h' are host states, modelling the relevant parts of the OS and network hardware of a machine, and l is of the following forms.

- msg for the host receiving a datagram msg from the network;
- $\overline{msg}$ for the host sending a datagram msg to the network;
- $tid{\cdot}\mathrm{f}(arg_1, .., arg_n)$ for a Sockets API call f made by thread tid, e.g. $tid{\cdot}\mathrm{bind}(fd, is_1, ps_1)$ for a bind() call with arguments (fd, is_1, ps_1) for the file descriptor, IP address, and port;
- $\overline{tid{\cdot}v}$ for value v being returned to thread tid by the Sockets API;
- τ for an internal transition by the host, e.g. for a datagram being taken from the host's input queue and processed, possibly enqueuing other datagrams for output; and
- d for time $d \in \mathbb{R}_{>0}$ passing.

In addition there are labels for loopback messages, changes of network interface status, and certain BSD debug trace events (these reveal some of the implementation internal state at specific points).

There are many careful choices embodied in the form of this definition, of exactly what aspects of the real system to model, which events to observe, and what (rather mild) abstraction and idealisation is done to map them to events in the model. We refer the reader to [5, 6] for discussion.

Internal structure The host LTS is defined in an operational semantics style as the least relation satisfying certain rules. These rules form the bulk of the specification: some 148 rules for the socket calls (5–10 for each interesting call) and some 46 rules for message send/receive and for internal behaviour, occupying around 160 and 75 pages respectively, including extensive commentary. The remainder consists of definitions of types, constants, and auxiliary functions, around 125 pages.

The definition is almost entirely flat, in two senses. First, most rules have no transition premises, the only exceptions being rules for time passage (the definition is factored into two relations, one without time passage and one with). Second, there is no parallel composition structure or synchronisation within a host; each rule can refer to any part of the host state as needed.

Host states and types HOL types can be constructed from type constructors, built-in or user-defined, of natural-number arities. We make extensive use of this type structure, especially products ($t\#t$), functions ($t \rightarrow t$), finite maps ($t \mapsto t$), labelled records ($(\!|fld_1 : t_1, fld_2 : t_2, ...|\!)$), options, lists, 32-bit integers, natural numbers, real numbers, and user-defined datatypes. HOL also supports ML-style polymorphism. The specification is only intentionally polymorphic in a few places, but type inference and checking is essential. The HOL datatype and inductive definition packages automatically prove various theorems for later use; during the course of the project we have had to improve these to handle the large types required.

Host states h are simply values of a certain carefully-designed HOL type. We give two fragments of this definition below, to indicate the style: the types of sockets and of the protocol-specific information contained in a TCP socket. A host h contains a finite map from socket identifiers to sockets together with much other data (its open file descriptions, queues of incoming and outgoing messages, and so on).

– details of a socket :

```
socket
 =(| fid : fid option; (* associated open file description if any *)
     sf : sockflags; (* socket flags *)
     is1 : ip option; (* local IP address if any *)
     ps1 : port option; (* local port if any *)
     is2 : ip option; (* remote IP address if any *)
     ps2 : port option; (* remote port if any *)
     es : error option; (* pending error if any *)
     cantsndmore : bool; (* output stream ends at end of send queue *)
     cantrcvmore : bool; (* input stream ends at end of receive
                            queue *)
     pr : protocol_info (* protocol-specific information *)
   |)
```

– protocol-specific information for a TCP socket :

```
tcp_socket
 =(| st : tcpstate; (*LISTEN, ESTABLISHED, TIME_WAIT, etc.*)
     cb : tcpcb; (*the 'TCP control block', a 44-field record of timers,
                   sequence numbers, the reassembly segment queue, etc.*)
     lis : socket_listen option; (*data for listening socket*)
     sndq : byte list; (*send queue*)
     sndurp : num option; (*send urgent pointer*)
     rcvq : byte list; (*receive queue*)
     rcvurp : num option; (* receive urgent pointer*)
     iobc : iobc (*out-of-band data and status*)
   |)
```

Higher-order types are used to a limited extent. For example, one auxiliary definition in the specification has the type below.

$$\mathsf{arch} \rightarrow (\mathsf{ifid} \mapsto \mathsf{ifd}) \rightarrow \mathsf{socket} \rightarrow (\mathsf{bool}\#(\mathsf{tcpcb} \rightarrow \mathsf{tcpcb})\ \mathsf{option})$$

and there are some 3rd-order types (where $\mathrm{order}(tycon) = 0$, $\mathrm{order}(t \rightarrow t') = \mathrm{order}(t \mapsto t') = \max(\mathrm{order}(t)+1, \mathrm{order}(t'))$, and $\mathrm{order}((t, t', ..)tyop) = \max(\mathrm{order}(t), \mathrm{order}(t'), ..)$. The order could be reduced by encoding finite maps with lists, at some loss of clarity, but in HOL there is no need to do so. The highest order at which we quantify is order 1, e.g. at the host type.

Transition rules Each transition rule is abstractly of the form

$$\vdash P \Rightarrow h_0 \xrightarrow{l} h$$

where P is a condition (on the free variables of h_0, l, and h) under which host state h_0 can make a transition labelled l to host state h. The condition is usually written below the transition. One of the simplest rules is shown below.

bind_5 **all: fast fail** **Fail with EINVAL: the socket is already bound to an address and does not support rebinding; or socket has been shutdown for writing on FreeBSD**

$$\frac{h\ (\!|ts := ts \oplus (tid \mapsto (\textsc{Run})_d)|\!)}{\xrightarrow{tid\cdot \mathrm{bind}(fd, is_1, ps_1)}\ h\ (\!|ts := ts \oplus (tid \mapsto (\textsc{Ret}(\text{FAIL EINVAL}))_{\mathrm{sched\_timer}})|\!)}$$

$$\begin{array}{l} fd \in \mathbf{dom}(h.fds) \wedge \\ fid = h.fds[fd] \wedge \\ h.files[fid] = \textsc{File}(\text{FT\_SOCKET}(sid), ff) \wedge \\ h.socks[sid] = sock \wedge \\ (sock.ps_1 \neq * \vee \\ (\mathrm{bsd\_arch}\ \ h.arch \wedge sock.pr = \text{TCP\_PROTO}(tcp\_sock) \wedge \\ \quad (sock.cantsndmore \vee \\ \quad tcp\_sock.cb.bsd\_cantconnect))) \end{array}$$

This is one of 7 rules for bind(). It deals with the case where a thread tid calls $\mathrm{bind}(fd, is_1, ps_1)$ for a socket referenced by the file descriptor fd that already has its local port bound; the error EINVAL will be returned to the thread. In the host on the left of the transition, the thread state map ts maps thread id tid to $(\textsc{Run})_d$, indicating that the thread is running (in particular, it is not currently engaged in a socket call). In the host on the right of the transition, that thread is mapped to $(\textsc{Ret}(\text{FAIL EINVAL}))_{\mathrm{sched\_timer}}$, indicating that within time sched_timer the failure EINVAL should be returned to the thread (all returns are handled by a single rule *return_1*, which generates labels $\overline{tid\cdot v}$).

The sidecondition is a conjunction of 5 clauses. The first three ensure that the file descriptor fd is in the host's file descriptor map $h.fds$, that fid is the file identifier for this file descriptor, and that this fid is mapped by the host's files map $h.files$ to $\textsc{File}(\text{FT\_SOCKET}(sid), ff)$, i.e. to a socket identifier sid and file flags ff. The fourth simply picks out the socket structure $sock$ associated with the socket id sid. The fifth says that the local port

$accept\_1$	**tcp: succeed**	Return new connection; either immediately or from a blocked state.
$accept\_2$	**tcp: block**	Block waiting for connection.
$accept\_3$	**tcp: fail**	Fail with EAGAIN: no pending connections and non-blocking semantics set.
$accept\_4$	**tcp: fail**	Fail with ECONNABORTED: the listening socket has $cantsndmore$ set or has become CLOSED. Returns either immediately or from a blocked state.
$accept\_5$	**tcp: fail**	Fail with EINVAL: socket not in LISTEN state.
$accept\_6$	**tcp: fail**	Fail with EMFILE: out of file descriptors.
$accept\_7$	**udp: fail**	Fail with EOPNOTSUPP or EINVAL: accept() called on a UDP socket.

Figure 1. The rules for the accept() Sockets API call.

of the socket with that sid is not equal to the wildcard $*$, i.e. that this socket has already got its local port bound, or that some BSD-specific condition holds.

Nondeterminacy and relational specification For the specification to actually include all the behaviour even of a single implementation it must be highly nondeterministic, e.g. to admit the pseudo-random choice of initial sequence numbers, the variations due to varying OS scheduling of multiple threads and interrupts, and variations in the rates of timers.

This nondeterminism forces us to use relational idioms (expressed in the higher-order logic of HOL) throughout much of the specification. In places we can use auxiliary functions, but often we need auxiliary relations, or functions that return relations. Nondeterminism is sometimes implicit (e.g. where several different error rules are applicable) and sometimes explicit (e.g. where an unconstrained or partially-constrained variable is introduced).

Nondeterminism is also used to model some differences between implementations (e.g. unconstraining the protocol options chosen at connection-establishment time). Other implementation differences are modelled by explicitly parameterising the behaviour by an implementation version (e.g. as in the last conjunct of $bind\_5$, which is BSD-specific). This explicitness lets us identify and test differences more sharply.

Often it is useful to think of a part of a rule predicate P as being a 'guard', which is a sufficient condition for the rule to be applicable, and the remainder as a constraint, which should always be satisfiable, on the final state h. This distinction is not formalised, however.

Imperative updates and the relational monad In the C code of the implementations the early parts of segment processing can have side-effects on the host data structures, especially on the TCP control block, before the outcome of processing is determined. Disentangling this imperative behaviour into a clear declarative specification is non-trivial. Our most complicated rule $deliver\_in\_3$ calculates the host's response to an incoming segment after a connection has been established. This rule makes use of a relational monad structure to expose certain intermediate states (as few as possible). Relations in this monad have (curried) types of the form

$$t \to t \# t' \# \mathsf{bool} \to \mathsf{bool}$$

where t is the state bring manipulated (e.g. a pair of a socket and a host's bandlimiter state), t' is the result type (e.g. a list of segments to be sent in reply to a segment being processed), and the boolean in the second tuple argument is a flag indicating whether or not execution should continue.

There is a binding combinator $\mathrm{andThen}$, a unit cont (which does nothing and continues), and a zero stop (which does nothing and stops), and various other operations to manipulate the state. It should be a theorem that $\mathrm{andThen}$ is associative, and so forth, though we have not checked this within HOL.

Time and urgency Much TCP behaviour is driven by timers and timeouts, and distributed applications generally depend on timeouts in order to cope with asynchronous communication and failure. Our model bounds the time behaviour of certain operations: for example, a failing bind call in $bind\_5$ will return after a scheduling delay of at most $\mathrm{dschedmax}$; a call to pselect with no file descriptors specified and a timeout of 30sec will return at some point in the interval $[30, 30 + \mathrm{dschedmax}]$ seconds. Some operations have both a lower and upper bound; some must happen immediately; and some have an upper bound but may occur arbitrarily quickly. For some of these requirements time is essential, and for others time conditions are simpler and more tractable than the corresponding fairness conditions [16, §2.2.2].

Time passage is modelled by transitions labelled $d \in \mathbb{R}_{>0}$ interleaved with other transitions, which are regarded as instantaneous. This models global time which passes uniformly for all parts of the system (although it cannot be accurately observed internally by any of them). States are defined as *urgent* if there is a discrete action which we want to occur immediately. This is modelled by *prohibiting* time passage steps d from (or through) an urgent state. We have carefully arranged the model to avoid pathological timestops by ensuring a local receptiveness property holds: the model can always perform input transitions for any label one might reasonably expect it to.

The model is constructed to satisfy the two time axioms of [16, §2.1]. Time is *additive*: if $h_1 \xrightarrow{d} h_2$ and $h_2 \xrightarrow{d'} h_3$ then $h_1 \xrightarrow{d+d'} h_3$; and time passage has a *trajectory*: roughly, if $h_1 \xrightarrow{d} h_2$ then there exists a function w on $[0, d]$ such that $w(0) = h_1$, $w(d) = h_2$, and for all intermediate points t, $h_1 \xrightarrow{t} w(t)$ and $w(t) \xrightarrow{d-t} h_2$. These axioms ensure that time passage behaves as one might expect.

The timing properties of the host are specified using a small range of *timers*, each with a particular behaviour. A single transition rule $epsilon\_1$ (shown in §3) models time passage, say of duration d, by evolving each timer in the model state forward by d. If any timer cannot progress this far, or the initial model state is marked as urgent for another reason, then the rule fails and the time passage transition is disallowed. Note that, by construction, the model state may only become urgent at the expiry of a timer or after a non-time-passage transition. This guarantees correctness of the above rule. The timers ensure that the specification models the behaviour of real systems with (boundedly) inaccurate clocks.

Many timed process algebras enforce a *maximal progress* property [29], requiring that any action (such as a CCS synchronisation) must be performed immediately it becomes enabled. We found this too inflexible for our purposes; we wish to specify the behaviour of, e.g., the OS scheduler only very loosely, and so it must be possible to nondeterministically delay an enabled action, but we did not want to introduce many nondeterministic choices of delays. Our calculus therefore does not have maximal progress; instead we ensure timeliness properties by means of timers and urgency. Our reasoning using the model so far involves only finite trace properties, so we do not need to impose Zeno conditions.

Partitioning the behaviour The partition of the system behaviour into particular rules is an important aspect of the specification. We have tried to make it as clear as possible: each rule deals with a conceptually different behaviour, separating (for example) the error cases from the non-error cases. This means there is some repetition of clauses between rules. For example, many rules have a predicate clause that checks that a file descriptor is legitimate. Individual rules correspond very roughly to execution *paths* through implementation code. For substantial aspects of behaviour, on the other hand, we try to ensure they are localised to one place in the specification. For example, calls such as accept() might have a successful return either immediately or from a blocked state. The final outcome is similar in both, and so we have a single rule ($accept\_1$) that deals with both cases. Another rule ($accept\_2$) deals with entering the blocked states, and several others with the various error cases. The various accept rules are summarised in Fig. 1.

Framing The host state is complex, but most rules need refer only to a small part of it, and permit an even smaller part to differ between the initial and final state of a transition. In designing the host state type and the rules it is important to ensure that explicit frame conditions are usually not needed, to avoid overwhelming visual noise. To do so we use a combination of pattern matching (in the h_0 and h) and of (functional) projection and update operations for records and finite maps.

The overall host state structure roughly follows that of the system state: hosts have collections of socket data structures, message input and output queues, etc.; sockets have local and remote IP addresses and ports, etc.; TCP sockets have a TCP control block, and so on. The details vary significantly, however, with our structures arranged for clarity rather than performance — as we are specifying only the externally-observable behaviour, we can choose the internal state structure freely. For example, TCP send and receive queues are modelled by byte lists rather than the complex BSD `mbuf` structures, and we can subdivide the state so that commonly-accessed components are together and near the root.

Concurrency, blocking calls, and atomicity In the implementations the host state may be modified by multiple threads making Sockets API calls (possibly for the same socket), and by OS interrupt handler code prompted by timers or incoming messages. Sockets API calls can be fast or slow, the latter potentially blocking for arbitrarily long. The imperative C code modifies state as it executes. Fortunately most of the network protocol code is guarded by a coarse-grained lock, so the specification need not consider all possible interleavings. Fast calls are typically modelled by two atomic transitions, one for the call, in which all state change happens (as in $bind\_5$), and one for the return of the result. Slow calls typically involve three transitions, one for the call (leaving the host thread record in a special blocked state), one in which the call is unblocked (e.g. a τ transition when new data is processed), and one for the return of the result. Applying a degree of fuzziness to times and deadlines suffices to let this correspond to the real executions.

Automated typesetting HOL source is fairly readable in the small, but good typesetting and clear large-scale structure are essential to make a large specification intelligible, and manual approaches would be tedious and error-prone. We have therefore built an automated typesetting system that takes HOL source and outputs LaTeX, essentially as in the parts quoted here. This has been surprisingly important in making the specification comprehensible, during development and for others.

3. Validation: Checking Technology

Our computational task is this: given the nondeterministic labelled transition system $\xrightarrow{l}$ of the host LTS, an initial host h_0, and a sequence of experimentally observed labels $l_1 \ldots l_n$, determine whether h_0 can exhibit this behaviour in the model. The transition system includes unobservable τ labels, so we actually have to demonstrate a sequence

$$h_0 \xrightarrow{\tau*} \xrightarrow{l_1} \xrightarrow{\tau*} \xrightarrow{l_2} \ldots \xrightarrow{\tau*} \xrightarrow{l_n} h$$

If the system were deterministic, the problem would be easily solved. The initial conditions are completely specified and the problem would be one of mechanical calculation with values that were always ground. Because the system is nondeterministic, the problem becomes one of exploring the tree of all possible traces that are consistent with the given label sequence. This exploration is not entirely label-driven, as it must consider the possibility that a τ transition is required.

Search structure Nondeterminism arises in two different ways:

- two or more rules may apply to the same host-label pair (or the host may be able to undergo a τ transition);
- a single rule's sideconditions may not constrain the resulting host to take on just one possible value.

These two sorts of nondeterminism do not correspond to any deep semantic difference, but do affect the way in which the problem is solved.

Because labels come in a small number of different categories, the number of rules that might apply to any given host-label pair is relatively small. It is clearly reasonable to explicitly model this nondeterminism by explicit branching within a tree-structured search-space. The search through this space is done depth-first.

Possible τ transitions are checked last: if considering host h and a sequence of future labels, and no normal rule allows for a successful trace, posit a τ transition at this point, followed by the same sequence of labels. As long as hosts can not make infinite sequences of τ transitions, the search-space remains finite.

An example of the second sort of nondeterminism comes when a resulting host is to include some numeric quantity, but where the model only constrains this number to fall within certain bounds. It is clearly foolish to explicitly model all these possibilities as branching (indeed, for many types there are an infinite number of possibilities). Instead, the system maintains sets of constraints (HOL predicates), attached to each transition. Instead of finding a sequence of theorems of the form

$$\begin{array}{l} \vdash h_0 \xrightarrow{l_1} h_1 \\ \vdash h_1 \xrightarrow{l_2} h_2 \\ \cdots \\ \vdash h_{n-1} \xrightarrow{l_n} h_n \end{array}$$

(eliding the τs now) we must find a sequence of theorems of the form

$$\begin{array}{l} \Gamma_0 \vdash h_0 \xrightarrow{l_1} h_1 \\ \Gamma_0 \cup \Gamma_1 \vdash h_1 \xrightarrow{l_2} h_2 \\ \cdots \\ \bigcup_{i=0}^{n-1} \Gamma_i \vdash h_{n-1} \xrightarrow{l_n} h_n \end{array}$$

where each Γ_i is the set of constraints generated by the i-th transition. If the fresh constraints were only generated because new components of output hosts were under-constrained, there would be no difficulty with this.

Unfortunately, the sideconditions associated with each rule will typically refer to input host component values that are no longer ground, but which are instead constrained by a constraint generated

by the action of an earlier rule. For example, imagine that the first transition of a trace has made the v component of the host have a value between 1 and 100. Now faced with an l-transition, the system must eliminate those rules which allow for that transition if v is greater than 150.

The symbolic evaluator accumulates constraint sets as a trace proceeds, and checks them for satisfiability. The satisfiability check takes the form of simplifying each assumption in turn, while assuming all of the other assumptions as context. HOL simplification includes the action of arithmetic decision procedures, so unsatisfiable arithmetic constraints are discovered as well as more obviously unsatisfiable constraint sets. For example, using the theorems proved by HOL's data type technology, the simplifier "knows" that the constructors for algebraic types are disjoint. Thus, $(s = [\,]) \wedge (s = h :: t)$ is impossible because the nil and cons constructors for lists are disjoint.

Constraint instantiation As a checking run proceeds, later labels may determine variables that had initially been under-determined. For example, Windows XP picks file descriptors for sockets non-deterministically, so on this architecture the specification for the socket call only requires that the new descriptor be fresh. As a trace proceeds, however, the actual descriptor value chosen will be revealed (a label or two later, the value will appear in the return-label that is passed back to the caller). In this situation, and others like it, the set of constraints attached to the relevant theorem will get smaller when the equality is everywhere eliminated. Though the checker does not explicitly do this step, the effect is as if the earlier theorems in the run had also been instantiated with the value chosen. If the value is clearly inconsistent with the initial constraints, then this will be detected because those constraints will have been inherited from the stage when they were generated.

Case splitting Sometimes a new constraint will be of a form where it is clear that it is equivalent to a disjunction of two possibilities. Then it often makes sense to case-split and consider each arm of the disjunction separately

$$\Gamma, p \vee q \vdash h_{i-1} \xrightarrow{l_i} h_i$$

$$\Gamma, p \vdash h_{i-1} \xrightarrow{l_i} h_i \qquad \Gamma, q \vdash h_{i-1} \xrightarrow{l_i} h_i$$

At the moment, such splitting is done on large disjunctions (as above), and large conditional expressions that appear in the output host. For example, if the current theorem is

$$\Gamma \vdash h_0 \xrightarrow{l} (\ldots \text{if } p \text{ then } e_1 \text{ else } e_2 \ldots)$$

then two new theorems are created: $\Gamma, p \vdash h_0 \xrightarrow{l} (\ldots e_1 \ldots)$ and $\Gamma, \neg p \vdash h_0 \xrightarrow{l} (\ldots e_2 \ldots)$, and both branches are explored (again, in a depth-first order).

The Core Algorithm: Evaluating One Transition Given a host h_0 (expressed as a set of bindings for the fields that make up a host, and thus of the form $\langle\!\langle \mathit{fld}_1 := v_1; \mathit{fld}_2 := v_2; ... \rangle\!\rangle$), a set of constraints Γ_0 over the free variables in h_0, and a ground label l_0 (whether from the experimentally observed trace, or a τ label), the core processing step of the trace-checking algorithm is to generate a list of all possible successor hosts, along with their accompanying constraints.

We precompute theorems of the form

$$\langle\!\langle \mathit{fld}_1 := v_1; \mathit{fld}_2 := v_2; ... \rangle\!\rangle \xrightarrow{l} h \equiv (D_1 \ \vee \cdots D_{n-1} \vee D_n) \quad (1)$$

where l is a label form (such as $(\mathit{tid}{\cdot}\text{socket}(\mathit{arg}))$ or τ or $\overline{\mathit{msg}}$) that will match l_0, and where each D_i corresponds to a rule in the definition of the transition system. Such a theorem can be matched against the input host and label. Each D_i will constrain both the input fields and the output host h. More, each D_i includes an equation of the form $h = \langle\!\langle \mathit{fld}_1 := v_1'; \mathit{fld}_2 := v_2'; ... \rangle\!\rangle$, where the new, primed variables are existentially quantified in D_i, and further constrained there.

It is then straightforward to generate a sequence of theorems (one per possible rule), each of the form

$$\vdash D_i \Rightarrow \langle\!\langle \mathit{fld}_1 := v_1; ... \rangle\!\rangle \xrightarrow{l_0} \langle\!\langle \mathit{fld}_1 := v_1'; ... \rangle\!\rangle$$

where any variables existentially quantified in D_i are now implicitly universally quantified in the theorem, and may appear in the consequent of the implication. Similarly, the process of matching the input values against the precomputed theorem (1) will have affected the form of D_i.

Now the initial context Γ_0 can be brought into play, and assumed while the D_i is simplified in that context. For example, we earlier discussed the scenario where a variable in the input host might become constrained in Γ_0 to be no larger than 100. If some D_k insists that the same value be greater than 150, the process of simplification will discover the contradiction, and rewrite this D_k to false. In such a scenario, the theorem containing D_k will become the vacuous $\Gamma_0 \vdash \top$, and can be discarded.

Those theorems that survive this stage of simplification can then be taken to the form

$$\Gamma_0, D_i' \vdash \langle\!\langle \mathit{fld}_1 := v_1; ... \rangle\!\rangle \xrightarrow{l_0} \langle\!\langle \mathit{fld}_1 := v_1'; ... \rangle\!\rangle$$

The next phase of evaluation is "context simplification". Though some checking and simplification of the constraints in D_i' has been performed, the constraints there have not yet caused any adjustment to Γ_0. In this phase, the implementation iterates through the hypotheses in $\Gamma_0 \cup D_i'$, simplifying each hypothesis in turn while assuming the others. Furthermore, if this process does induce a change in the hypotheses, the process is restarted so that the new form of the hypothesis is given a chance to simplify all of the other members of the set.

After the first phase of context simplification, the checker heuristically decides on possible case-splits. If a case-split occurs, more context simplification is required because the new hypothesis in each branch will likely induce more simplification.

This phase of evaluation is potentially extremely expensive. We have made various improvements to the checker during development that have made dramatic differences, but they do not reflect any deep theoretical advances. Rather, we are engaged in "logic engineering" on top of the HOL kernel. The LCF philosophy of the latter means that the ad-hoc nature of parts of our implementation cannot affect soundness. At worst we will harm the completeness of a method already known to be essentially incomplete because of the undecidability of the basic logic. In fact, incompleteness is pragmatically less important than being able to quickly reduce formula sizes, and to draw inferences that will help in subsequent steps.

Laziness in Symbolic Evaluation Because hosts quickly lose their groundedness as a checking run proceeds, many of the values being computed with are actually constrained variables. Such variables may even come to be equated with other expressions, where those expressions in turn include unground components. It is important in this setting to retain variable bindings rather than simply substituting them out. Substituting unground expressions through large terms may result in many instances of the same, expensive computation when those expressions do eventually become ground.

This is analogous to the way in which a lazy language keeps pending computations hidden in a "thunk" and does not evaluate them prematurely. The difference is that lazy languages "force"

thunks when evaluation determines that their values are required. In the trace-checking setting, expressions yield values as the logical context becomes richer, not on the basis of whether or not those values are required elsewhere.

Moreover, as soon as an expression yields up a little information about its structure it is important to let this information flow into the rest of the formula. For example, if the current theorem is

$$x = E \vdash \ldots (\mathsf{if}\ x = [\,]\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x)) \ldots$$

then it is important not to substitute E for x and end up working with two copies of (presumably complicated) expression E. On the other hand, future work may reveal that E is actually of the form $h :: t$ for some (themselves complicated) expressions h and t.

In this case, the theorem must become

$$v_1 = h, v_2 = t \vdash \ldots (g(v_1 :: v_2)) \ldots$$

In this situation, the application of g to a list known to a be a cons-cell may lead to future useful simplification.

To implement this, the checker can isolate equalities to prevent them from being instantiated, and detects when expressions become value-forms, or partial value-forms.

Evaluating Time Transitions Time transitions require special treatment. An experimentally-observed trace will typically have a time passage transition, labelled with a duration, between each other observable transition. The relevant rule is

epsilon_1 **all: misc nonurgent** **Time passes**

$h \xrightarrow{dur} h'$

let $hs' = \text{Time\_Pass\_host}\ dur\ h$ **in**
is_some $hs' \wedge$
$h' \in (\textbf{the}\ hs') \wedge$
$\neg(\exists rn\ rp\ rc\ lbl\ h'.rn/ * rp, rc * /h \xrightarrow{lbl} h' \wedge \text{is\_urgent}\ rc)$

The rule says that host h can have its internal timers updated by the duration dur to become host state h', if h is not an *urgent* state. A host state is urgent if it is able to undergo a τ transition of any rule annotated **urgent** (this condition is expressed in the last line of the displayed rule). Such transitions represent actions that are held to happen instantaneously, and which must "fire" before any time elapses, e.g. the expiry of a pselect() timeout (plus a scheduling delay).

The trace-checker does not check for non-urgency by actually trying all of the urgent rules in turn. Instead, it uses a theorem (proved once and for all as the system builds) that provides an approximate characterisation of non-urgency. If this is satisfied, the above rule's sideconditions can be discharged, and progress made. If the approximation can not be proved true, then a τ step is attempted so that the host can move through its pending urgent transition.

Model Translation An important aim of the formalisation has been to support the use of a natural, mathematical idiom in the writing of the specification. This does not always produce logical formulas well-suited to automatic analyses. Even making sure that the conjuncts of a sidecondition are "evaluated" (simplified) in a suitable order can make a big difference to the efficiency of the tool. Rather than force the specification authors and readership to deal explicitly with algorithmic issues (and the specification to be a Prolog-like program), we have developed a variety of tools to automatically translate a variety of idioms into equivalent forms.

At their best, these translations are produced by ML code written to handle an infinite family of possibilities. Written within HOL, this ML code produces translations by proving logical equivalences. In this way, we can be sure that the translation is correct, i.e. that the semantics of the specification is preserved. In other cases, we prove specific theorems that state a particular rule or auxiliary function is equivalent to an alternative form. This theorem then justifies the use of the more efficient expression of the same semantics.

Translating Non-injective Pattern-Matching One important example of translation comes in the handling of the pattern-matching idiom. Making use of the HOL syntax for record values updated at specific fields, specifiers can write

$$h\ \langle\!|\mathit{fld}_1 := v_0|\!\rangle \xrightarrow{l} h\ \langle\!|\mathit{fld}_1 := v|\!\rangle$$

to adjust the host h. The problem with field updates is that they are not injective functions: there are multiple instantiations for h given any particular host meant to match this rule. The transformation in this case is simple: h is expanded into a complete listing of all its possible fields, the actions of the update functions are applied, and the translated rule becomes

$$\langle\!|\mathit{fld}_1 := v_0; \mathit{fld}_2 := v_2; \mathit{fld}_3 := v_3; ...|\!\rangle \xrightarrow{l} \langle\!|\mathit{fld}_1 := v; \mathit{fld}_2 := v_2; \mathit{fld}_3 := v_3; ...|\!\rangle$$

The specifier does not have to list the frame conditions, but the implementation of the evaluator is simplified by explicitly listing all of the fields (unchanging or not) in the transformed form.

Another example of non-injective pattern-matching comes with the use of finite maps. These values are manipulated throughout the labelled transition system. For example, rules describing the host's response to a new system call typically check that the calling thread is in the RUN state, and also specify the new state that the thread moves to if the transition is successful. Such a rule has the general form

$$\langle\!|ts := tidmap \oplus (t, \text{RUN}); ...|\!\rangle \xrightarrow{l} \langle\!|ts := tidmap \oplus (t, newstate); ...|\!\rangle$$

$$sideconditions$$

where the ts field of the host is a finite map from thread-identifiers to thread state information. A naïve approach to the symbolic evaluation of such a rule would attempt to find a binding for the variable $tidmap$. Unfortunately, in the absence of further constraints on that variable in the rule's sideconditions, there are multiple such bindings: $tidmap$ may or may not include another binding for the key t, and if it does include such a binding, may map t to any possible value. Because the only occurrences of $tidmap$ are in situations where an overriding value for t is provided, these possibilities are irrelevant, and the evaluator should not distract itself by looking for such values.

We have written ML code to check rules are of suitable form and to then translate the above into

$$\langle\!|ts := tidmap; ...|\!\rangle \xrightarrow{l} \langle\!|ts := tidmap \oplus (t, newstate)|\!\rangle$$

$$\text{fmscan}\ \ tidmap\ t\ \text{RUN} \wedge sideconditions$$

where the fmscan relation checks to see if its first argument (a finite map) maps its second argument to its third. It is characterised by the following theorem

$$\begin{array}{rcl} \mathrm{fmscan}\ \emptyset\ k_2\ v_2 & = & \bot \\ \mathrm{fmscan}\ (fm \oplus (k_1, v_1))\ k_2\ v_2 & = & (k_1 = k_2 \wedge v_1 = v_2) \vee \\ & & \mathrm{fmscan}\ (fm \backslash\backslash k_1)\ k_2\ v_2 \end{array}$$

where $fm \backslash\backslash k$ denotes the finite map that is equal to fm, except that any binding for k has been removed.

In other circumstances, the underlying finite map may not always appear with a suitable rebinding of the relevant key. For example, this happens in rules that remove key-value pairs from maps. Such a rule is $close\_7$, which models the successful closing of the last file-descriptor associated with a socket in the CLOSED, SYN_SENT or SYN_RECEIVED states. The rule's transition removes the socket-id/socket binding from the host's $socks$ map. The relevant parts of the rule look like

$$\begin{array}{l} \quad \langle\!\langle socks := sockmap \oplus (sid, sock); ... \rangle\!\rangle \\ \xrightarrow{tid \cdot \mathrm{close}(fd)} \\ \quad \langle\!\langle socks := sockmap; ... \rangle\!\rangle \end{array}$$

sideconditions (linking sid to fd, among other things)

Here the translation to the non-pattern version of the code can only succeed if the sideconditions include the fact that sid does not occur in the domain of the map $socks$. Without such a sidecondition, the meaning of the rule would be to allow the finite map to take on any possible binding for sid in the resulting state. Not including such a sidecondition is such an easy mistake for the specification-writer to make that the code implementing this transformation issues a warning if it can not find it.

If this constraint is found in the sideconditions, then the rule becomes

$$\begin{array}{l} \quad \langle\!\langle socks := sockmap; ... \rangle\!\rangle \\ \xrightarrow{tid \cdot \mathrm{close}(fd)} \\ \quad \langle\!\langle socks := sockmap \backslash\backslash sid; ... \rangle\!\rangle \end{array}$$

$$\begin{array}{l} \mathrm{fmscan}\ sockmap\ sid\ sock\ \wedge \\ sideconditions[sockmap := sockmap \backslash\backslash sid] \end{array}$$

where the sideconditions to the rule have acquired a new fmscan constraint, and have been altered so that any old references to $sockmap$ are replaced by $sockmap \backslash\backslash sid$.

Other Translation Examples A number of the specification's auxiliary functions are defined in ways that, while suitable for human consumption, are not so easy to evaluate. One simple example is the definition of a host's local IP addresses. Given a finite map from interface identifiers to interface data values, the function $\mathrm{local\_ips}$ is defined:

$$\mathrm{local\_ips}(ifmap) = \bigcup_{(k,i) \in ifmap} i.ipset$$

This definition is inconvenient to work with directly, so the equivalent recursive characterisation

$$\begin{array}{l} \mathrm{local\_ips}(\emptyset) = \emptyset \\ \mathrm{local\_ips}(ifmap \oplus (k, i)) = i.ipset \cup \mathrm{local\_ips}(ifmap \backslash\backslash k) \end{array}$$

is used instead.

Other translations rewrite definitions of relations to take on a prenex-form:

$$\begin{array}{rl} R\ x\ y = & \exists \vec{v}.\ \mathsf{let}\ u_1 = e_1\ \mathsf{in} \\ & \quad \mathsf{let}\ u_2 = e_2\ \mathsf{in} \\ & \quad\quad \cdots \\ & \quad\quad c_1 \wedge c_2 \wedge c_3 \wedge \ldots c_n \end{array}$$

The simplification strategy chosen by the checker could effect this transformation at run-time but there is no reason not to precompute it, and use the translated form of the definition instead of the original.

One of the specification's most complicated auxiliary definitions is that for reassembly of TCP data that has arrived out of order, characterised by the function $\mathrm{tcp\_reass}$. Involving two gruesome set-comprehensions, $\mathrm{tcp\_reass}$'s definition calculates the set of all possible valid reassemblies of a set of received segments. The theorem giving the alternative characterisation instead uses analogues of fold and map, making evaluation over concrete data much easier. (The data is concrete because it is from observed labels corresponding to the arrival of packets.)

Adding Constraints It is always *sound* to add fresh assumptions to a theorem. The following is a rule of inference in HOL:

$$\frac{\Gamma \vdash t}{\Gamma, p \vdash t}$$

Adding arbitrary constraints in this way may allow heuristic knowledge to be added, and thus used to guide the search for a satisfying path. To date, we have not attempted to do this. The risk of such an activity is not unsoundness, but rather incompleteness: if we add an assertion p, and then find that this produces an unsatisfiable set of constraints, we may incorrectly conclude that there is no satisfying path.

On the other hand, we *do* add constraints that are consequences of existing assumptions. This preserves satisfiability. For example, traces often produce rather complicated expressions about which arithmetic decision procedures can not reason directly. We help the procedures draw conclusions by separately inferring upper and lower bounds information about such expressions, and adding these new (but redundant) assumptions to the theorem.

Simplification The core logical operation of the trace-checker is *simplification*. This can be characterised as term-rewriting with equational theorems, augmented with the use of various decision procedures.

The equational theorems used in rewriting may include sideconditions. The simplifier will try to discharge such by simplifying them to truth. If this succeeds, the rewrite can be applied. For example, integer division and modulus have no specified value when the divisor is zero, meaning that theorems about these constants are typically accompanied by sideconditions requiring the divisor to be non-zero.

This basic term-rewriting functionality is then augmented with decision procedures, such as those for Presburger arithmetic over $\mathbb{R}$ and $\mathbb{N}$, whose action is intermingled with rewriting. (Decision procedures typically rewrite sub-terms to $\top$ or $\bot$.) This use of (augmented) rewriting is well-established in the interactive theorem-proving community. Systems such as ACL2 have long provided such facilities, and demonstrated the potency of the combination. Equational rewriting provides an easy way to express core identities in theories that may or may not be decidable. Well-chosen identities can rapidly extend a system to cover significant parts of new theories.

In our non-interactive setting, it is additionally important to be able to add bespoke reasoning procedures to the action of the

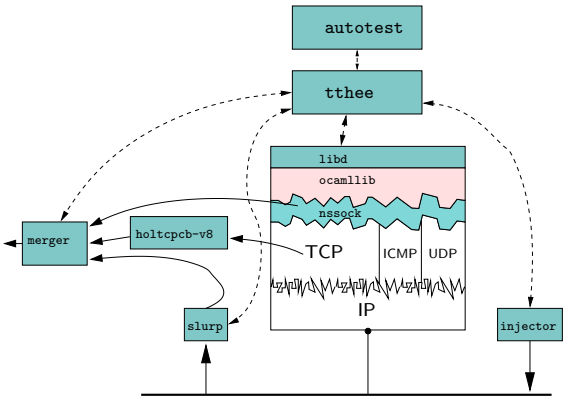


Figure 2. Testing infrastructure: a sample configuration.

simplifier. Our system extends the basic HOL simplifier not just with new rewrites, but also with new code, which handles cases not easily treatable with rewrites. Such programmatic extensions can not compromise the system's soundness because the programming is over the HOL kernel, which remains unchanged.

In addition to extensions already discussed, such as the lazy treatment of variable bindings, another example of such an extension is the treatment of TCP's 32-bit sequence numbers. For the most part, these are operated on as are normal fixed-width integers (with wrap-around arithmetic). For example, subtraction is such that $1 - 2 = -1$, but $-(2^{31}) - 3 = 2^{31} - 3$. The orderings on sequence numbers are defined to be

$$s_1 \lhd s_2 \equiv s_1 - s_2 \lhd 0$$

where $\lhd$ is any of $\{<, \leq, >, \geq\}$, and where the subtraction on the right results in an integer, which is compared with 0 in the normal way for $\mathbb{Z}$. These orderings exhibit some odd behaviours. For example, $s_1 < s_2 \not\equiv s_2 > s_1$ (consider s_1 and s_2 exactly 2^{31} apart).

Our system includes custom code for reasoning about inequalities on sequence numbers. This code is not a complete decision procedure, but has proved adequate in the checking runs performed to date.

Phasing In the early stages of the core algorithm, the priority for simplification is to eliminate possible transition rules that clearly can not apply. Checking rules' preliminary guards, and quickly eliminating those that are false is vital for efficiency. At this stage, therefore, it is wise not to expand the definitions of the more complicated auxiliaries. Such expansions would dramatically increase term-sizes, but might end up being discarded when a rule's guards were found not to be satisfied.

To implement this, we phase our use of the simplifier, so that it only simplifies with the simplest definitions early on. In this way, we hope to only expand complicated auxiliaries when they have a reasonable chance of being needed.

4. Validation: Experiments and Results

Experiments To generate traces of the real-world implementations in a controlled environment we set up an isolated test network, with machines running each of our three OS versions, and wrote instrumentation and test generation tools. A sample test configuration is illustrated in Fig. 2. A test executive `tthee` drives the system by making Sockets API calls (via a `libd` daemon) and directly injecting messages with an `injector` tool. The resulting traces are HOL-parsable text files containing an initial host state (its interfaces, routing table, etc.), an initial time, and a list of timestamped labels.

Tests are scripted above `tthee`. They are of two kinds. The most straightforward use two machines, one instrumented and an auxiliary used as a communication partner, with socket calls invoked remotely. The others use a virtual auxiliary host, directly injecting messages into the network; this permits tests that are not easily produced via the Sockets layer, e.g. with re-ordering, loss, or illegal or nonsense segments. We have written tests to, as far as we could, exercise all the interesting behaviour of the protocols and API. Almost all tests are run on all three OSs; many are automatically iterated over a selection of TCP socket states, port values, etc.

Checking these traces, which had to be repeated often during development of the specification, is computationally intensive but naturally parallel. Using around 100 processors, checking 2600 UDP traces takes approximately 5 hours; checking 1100 TCP traces (for BSD only) takes approximately 50 hours. Analysing the results is done with several HTML and graphical visualisation tools.

Results The experimental validation process shows that the specification admits almost all the test traces generated. For UDP, over all three implementations (BSD, Linux, and WinXP), 2526 (97.04%) of 2603 traces succeed. For TCP we have focussed recently on the BSD traces, and here 1004 (91.7%) of 1095 traces succeed.

While we have not reached 100% validation, we believe these figures indicate that the model is for most purposes very accurate — certainly good enough for it to be a useful reference. Further, we believe that closing the gap would only be a matter of additional labour, fixing sundry very local issues rather than needing any fundamental change to the specification or the tools.

Of the TCP non-successes: 42 are due to checker problems (mainly memory limits); 6 are due to problems in test generation; and the remaining 43 traces are due to a collection of 20 issues in the specification which we have roughly diagnosed but not yet fixed.

The success rates above are only meaningful if the generated traces do give reasonable coverage. Care was taken in the design of the test suite to cover interesting and corner cases, and we can show that almost all rules of the model are exercised in successful trace checking. Of the 194 host LTS rules 142 are covered in at least one successful trace check run; 32 could not be covered by the tests (e.g. rules dealing with file-descriptor resource limits, or non-BSD TCP behaviour); and 20 either have not had tests written or have not yet succeeded in validation.

The goal of this project was not to find bugs in the implementations. Indeed, from a post-hoc specification point of view, the implementation behaviour, however strange, is a de facto standard which users of the protocols and API should be aware of. Moreover, to make validation of the specification against the implementation behaviour possible, it must include whatever that behaviour is. Nonetheless, in the course of the work we have found many behavioural anomalies (around 30 are listed in [6]), some of which are certainly bugs in the conventional sense. All are relatively local issues — the implementations are extremely widely used, so it would be very surprising to find serious problems in the common-case paths.

For example, in BSD:

- The receive window is updated on receipt of a bad segment.
- Simultaneous open can respond with an ACK rather than a SYN,ACK.
- There is 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 lead to pathological segments being transmitted, with no flags or only a FIN.
- After repeated retransmission timeouts the RTT estimates are incorrectly updated.
- 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 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.
- On Linux, options can be sent in a SYN,ACK that were not in the received SYN.

There are explicit OS version dependencies on around 260 lines of the specification. Many of these, and many of the bugs above, were discovered by our testing process, and indicate how discriminating it is. The remainder were found directly in the source code while writing the specification.

5. Related Work

There is an extensive literature dealing with semantic properties of deployed code, including work on software model-checking and on formal reasoning techniques, another (largely disjoint) literature on protocol specification and verification, and yet another on automated reasoning tools such as HOL. To the best of our knowledge, however, no previous work deals rigorously with behavioural properties comparable to those of real-world TCP.

The most detailed rigorous specification of a TCP-like protocol we are aware of is that of Smith [23], an I/O automata specification and implementation, with a proof that one satisfies the other, used as a basis for work on T/TCP. The protocol is still substantially idealised, however: congestion control is not covered, nor are options, and the work supposes a fixed client/server directionality. Later work by Smith and Ramakrishnan uses a similar model to verify properties of a model of SACK [22].

Musuvathi and Engler have applied their CMC model-checker to a Linux TCP/IP stack [17]. The properties checked were of two kinds: resource leaks and invalid memory accesses, and protocol-specific properties specified by a hand translation of the RFC793 state diagram into C code. While this is a useful model of the protocol, it is an extremely abstract view, omitting flow control, congestion control etc. Four bugs in the Linux implementation were found.

Bhargavan *et al.* develop an automata-theoretic approach for monitoring of network protocol implementations, with classes of properties that can be efficiently checked on-line in the presence of network effects [2]. They show that certain properties of TCP implementations can be expressed.

In a rare application of rigorous techniques to actual standards, Bhargavan, Obradovic, and Gunter use a combination of the HOL proof assistant and the SPIN model checker to study properties of distance-vector routing protocols [3], proving correctness theorems. In contrast to our experience for TCP, they found that for RIP the existing RFC standards were precise enough to support "without significant supplementation, a detailed proof of correctness in terms of invariants referenced in the specification". The protocols are significantly simpler: their model of RIP is (by a naïve line count) around 50 times smaller than the specification we present.

Alur and Wang address the PPP and DHCP protocols [1]. For each they check refinements between models that are manually extracted from the RFC specification and from an implementation.

Many authors have considered radically idealised versions of network protocols; we refer the reader to [5, 6] for some representative examples.

6. Conclusion

Summary We have described novel techniques that have enabled us to deal rigorously with the behaviour of complex real-world software infrastructure: the TCP/IP and Sockets API implementations that form the de facto standard for most networked computation. Using them, we have developed a post-hoc specification of TCP, UDP, and Sockets that is rigorous, detailed, has broad coverage, is remarkably accurate, and yet remains readable.

We achieved *rigor* by writing our specification in operational semantics within mechanised higher-order logic. This choice of logic also enabled us to keep the specification readable: we have been able to write in a natural mathematical idiom, defining appropriate abstractions where necessary. For example, our relational monad allows a clean presentation of a complex and branching sequence of updates to a shared state, our use of real-timed transition systems and timers allows realistically-loose timing properties to be precisely specified, the HOL type structure has permitted a clean model of endpoint states, and nondeterminism lets the specification be sufficiently loose to admit a range of acceptable behaviour. At a higher level, we have organised the specification modularly, categorising transition rules to make the "big picture" apparent.

We achieved *accuracy* by writing a special-purpose symbolic model checker on top of the HOL system, using theorem-proving techniques to perform symbolic evaluation while retaining LCF-style confidence in its soundness. This trace-checker allowed us to mechanically verify that our specification captures behaviours exhibited by the three implementations studied, with over 3 000 observed traces checked. Our experimental semantics approach, describing the existing implementations as they are, has also led us to the discovery of many behaviours in those implementations that are bugs.

Our techniques might loosely be described as logic engineering: we have developed an approach that is pragmatically feasible even for this highly-challenging problem, dealing with the behaviour of many thousands of lines of multi-threaded and time-dependent C code that were (emphatically) not written with verification in mind.

The total effort required for the project, including our earliest experiments with UDP [20, 21, 26, 18], has been around 9 man-years over 3.5 calendar years, of which much has been devoted to idiom and tool development. This is substantial, and might not be well-motivated for software that is not in critical use, but is rather modest compared with the effort devoted over the last 25 years (and perhaps the next) to implementing and understanding these protocols and their API. Network protocols are an area in which behavioural standards are essential, as many implementations must interoperate.

Future Work There are many possible directions for future work.

Using the model The model can be used directly in several ways: informally as a reference work; as a basis for testing other implementations (perhaps requiring additional work to remove artifacts specific to the three we have based it on); as a basis for formal reasoning; as a setting in which to describe modifications to (e.g.) TCP's congestion control algorithms; and even, given refinement of some of the nondeterministic choices, as an executable prototype. Compton [9] has demonstrated fully formal reasoning about executable distributed OCaml code above our earlier UDP specification, including a network model and semantics for an OCaml fragment, using the Isabelle proof assistant [14]. Formal work above our TCP specification would require mastering a greater level of detail, but may still be practical. Li and Zdancewic are currently producing a Haskell implementation of TCP closely based on the specification, using efficient purely-functional datastructures to achieve good performance. Their work has uncovered a few bugs in the

specification that our testing did not, indicating that better coverage (and better tools for assessing coverage) are still needed.

Our testing has only imposed a lower bound on the set of traces of the model. Short of a protocol correctness proof, there is no obvious precise upper bound, but only an informal one of utility – in any of the above senses.

Returning to the code With our specification in hand, it would be interesting to return to the implementation code with predicate-abstraction or abstract-interpretation techniques, perhaps identifying internal properties of particular sequential C functions (e.g. `tcp_output.c`) that could be mechanically verified.

Stream-level specification Our specification deals with individual TCP segments on the wire interface rather than the stream abstraction provided by TCP that application programmers would usually reason in terms of. We are now writing a companion specification at that more abstract level (and for stereotyped, "known-to-be-safe" usage). This will explicitly reify the notion of stream, and abstract away from some of the details in the existing specification. Our existing validation technology should ensure the stream-level specification's accuracy.

Design-time specification and specification-based testing In the long term, the most significant line of work may be to carry out similar specification and conformance-testing work at design-time for new protocols and other infrastructure. Early rigorous specification could provide conceptual clarity, and a design-for-test approach (especially, making any internal nondeterminism directly observable), could make specification-based testing commonplace.

Acknowledgements We acknowledge support from a Royal Society University Research Fellowship (Sewell), a St Catharine's College Heller Research Fellowship (Wansbrough), EPSRC grant GR/N24872 *Wide-area programming: Language, Semantics and Infrastructure Design*, EPSRC grant EP/C510712 *NETSEM: Rigorous Semantics for Real Systems*, EC FET-GC project IST-2001-33234 PEPITO *Peer-to-Peer Computing: Implementation and Theory*, CMI UROP internship support (Smith), and EC Thematic Network IST-2001-38957 APPSEM 2. National ICT Australia is funded by the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

References

[1] R. Alur and B.-Y. Wang. Verifying network protocol implementations by symbolic refinement checking. In *Proc. CAV '01, LNCS 2102*, pages 169–181, 2001.

[2] K. Bhargavan, S. Chandra, P. J. McCann, and C. A. Gunter. What packets may come: automata for network monitoring. In *Proc. POPL*, pages 206–219, 2001.

[3] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.

[4] E. Biagioni. A structured TCP in Standard ML. In *Proc. SIGCOMM '94*, pages 36–45, 1994.

[5] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proc. SIGCOMM 2005 (Philadelphia)*, Aug. 2005.

[6] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview. Technical Report UCAM-CL-TR-624, Computer Laboratory, University of Cambridge, Mar. 2005. 88pp. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.

[7] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The specification. Technical Report UCAM-CL-TR-625, Computer Laboratory, University of Cambridge, Mar. 2005. 386pp. Available at `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.

[8] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Trans. Netw.*, 5(4):514–524, 1997. Full version of a paper in SIGCOMM '96.

[9] M. Compton. Stenning's protocol implemented in UDP and verified in Isabelle. In *Proc. 11th CATS, Computing: The Australasian Theory Symposium*, pages 21–30, 2005.

[10] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF, LNCS 78*. 1979.

[11] M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment*. Cambridge University Press, 1993.

[12] The HOL 4 system, Kananaskis-3 release. `hol.sourceforge.net`.

[13] IEEE and The Open Group. *IEEE Std 1003.1™-2001 Standard for Information Technology — Portable Operating System Interface (POSIX®)*. Dec. 2001. Issue 6. Available `http://www.opengroup.org/onlinepubs/007904975/toc.htm`.

[14] The Isabelle proof assistant. `http://isabelle.in.tum.de/`.

[15] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the Prolac protocol language. In *Proc. SIGGCOMM '99*, pages 3–13, August 1999.

[16] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[17] M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *Proc.NSDI: 1st Symposium on Networked Systems Design and Implementation*, pages 155–168, 2004.

[18] M. Norrish, P. Sewell, and K. Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *Proceedings of the 10th ACM SIGOPS European Workshop (Saint-Emilion)*, pages 49–53, Sept. 2002.

[19] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proc. SIGCOMM '97*, pages 167–179, 1997.

[20] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. Technical Report 515, Computer Laboratory, University of Cambridge, July 2001.

[21] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proc. TACS 2001: Fourth International Symposium on Theoretical Aspects of Computer Software, Tohoku University, Sendai*, Oct. 2001.

[22] M. A. Smith and K. K. Ramakrishnan. Formal specification and verification of safety and performance of TCP selective acknowledgment. *IEEE/ACM Trans. Netw.*, 10(2):193–207, 2002.

[23] M. A. S. Smith. Formal verification of communication protocols. In *Proc. FORTE IX/PSTV XVI*, pages 129–144, 1996.

[24] W. R. Stevens. *TCP/IP Illustrated Vol. 1: The Protocols*. 1994.

[25] W. R. Stevens. *UNIX Network Programming Vol. 1: Networking APIs: Sockets and XTI*. Second edition, 1998.

[26] K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proc. ESOP, LNCS 2305*, pages 278–294, Apr. 2002.

[27] G. K. Wright and W. R. Stevens. TCP state transition diagram. In *TCP/IP Illustrated, Volume 2: The Implementation* `http://www.kohala.com/start/pocketguide1.ps`.

[28] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated Vol. 2: The Implementation*. 1995.

[29] W. Yi. CCS + time = an interleaving model for real time systems. In *Proc. ICALP*, pages 217–228, 1991.