

Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API

DRAFT

STEVE BISHOP, University of Cambridge¹
MATTHEW FAIRBAIRN, University of Cambridge¹
MICHAEL NORRISH, NICTA, Canberra
TOM RIDGE, University of Leicester
PETER SEWELL, University of Cambridge
MICHAEL SMITH, University of Cambridge¹
KEITH WANSBROUGH, University of Cambridge¹

The TCP/IP protocols and Sockets API have an interesting status: they play a vital role in modern communication and computation, and interoperability between implementations is essential. But what exactly they are is surprisingly obscure: their original development focussed on “rough consensus and running code”, augmented by prose RFC specifications that do not precisely define what it means for an implementation to be correct. Ultimately, the actual standard is the de facto one of the common implementations, including, for example, the 15 000–20 000 lines of the BSD implementation — optimised and multithreaded C code, time-dependent, with asynchronous event handlers, intertwined with the operating system, and security-critical.

This paper reports on work done in the *Netssem* project to develop lightweight mathematically rigorous techniques that can be applied to such systems: to specify their behaviour precisely (but loosely enough to permit the required implementation variation) and to test whether these specifications and the implementations correspond. We developed post-hoc specifications of TCP, UDP, and Sockets, both of the service that they provide to applications (in terms of TCP bidirectional stream connections), and of the internal operation of the protocol (in terms of TCP segments and UDP datagrams), together with an abstraction function relating the two. These specifications are rigorous, detailed, readable, with broad coverage, and are rather accurate. Working within a general-purpose proof assistant (HOL4), we developed *language idioms* (within higher-order logic) in which to write the specifications: operational semantics with nondeterminism, time, system calls, monadic relational programming, etc. We followed an *experimental semantics* approach, validating the specifications 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 HOL4.

Having demonstrated that our logic-based engineering techniques suffice for handling real-world protocols, we argue that similar techniques could be applied to future critical software infrastructure at design time, leading to cleaner designs and (via specification-based testing) more robust and predictable implementations. In cases where specification looseness can be controlled, this should be possible with lightweight techniques, without the need for a general-purpose proof assistant, at relatively little cost.

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.

Additional Key Words and Phrases: Network Protocols, TCP/IP, Sockets, API, Specification, Conformance Testing, Higher-order Logic, HOL, Operational Semantics

We acknowledge support from EPSRC Programme Grant EP/K008528/1 *REMS: Rigorous Engineering for Mainstream Systems*, EPSRC Leadership Fellowship EP/H005633 (Sewell), 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. NICTA was funded by the Australian Government’s *Backing Australia’s Ability* initiative, in part through the Australian Research Council.

¹ when this work was done

Contents

1	Introduction	4
1.1	Standard Industry Practice	4
1.2	Specifications that are Executable as Test Oracles	5
1.3	De Facto Standards	5
1.4	The TCP/IP Network Protocols and Sockets API	6
1.5	Our Specifications	7
1.5.1	A Protocol-level Specification of the De Facto Standard	7
1.5.2	A Service-level Specification and Validated Abstraction Function	9
1.6	Technical Challenge: Specification Looseness	10
1.7	Technical Challenge: Relating Model and System	11
1.8	Non-goals, Non-approaches, and Limitations	13
1.9	Project History	15
1.10	Paper Structure	15
2	Background: TCP/IP, the Sockets API, and the Existing Specifications	15
2.1	The TCP/IP Protocols	15
2.2	The Sockets API	16
2.3	Standard Practice: Protocol and API Descriptions	17
3	Protocol-Level Specification	19
3.1	The External Form and Scope of the Specification	20
3.1.1	The Top-level Operational Semantics Judgement	20
3.1.2	Protocol Issues	20
3.1.3	Network Interface Issues	21
3.1.4	Sockets Interface Issues	21
3.1.5	The TCP Control Block and TCP_DEBUG Interface Issues	25
3.2	The Internal Structure of the Specification	25
3.2.1	Internal Structure: Statics (Host States and Types)	25
3.2.2	Internal Structure: Dynamics (Host Transition Rules)	26
3.2.3	Sample Protocol-level Transition Rule: <i>bind_5</i>	27
3.2.4	Sample Protocol-level Transition Rule: <i>send_3</i>	27
3.2.5	Sample Protocol-level Transition Rule: <i>deliver_in_1</i>	30
3.3	Specification Idioms	32
3.3.1	Nondeterminacy and Relational Specification	32
3.3.2	Imperative Updates and the Relational Monad	32
3.3.3	Time and Urgency	32
3.3.4	Partitioning the Behaviour	34
3.3.5	Relationship Between Code and Specification Structure	34
3.3.6	Framing	34
3.3.7	Concurrency, Blocking Calls, and Atomicity	35
3.3.8	Presentation and Automated Typesetting	35
3.4	Example Trace	35
4	Service-level Specification	36
4.1	Streams	37
4.2	Sample Service-level Transition Rule: <i>send_3</i>	38
5	Abstraction Function from Protocol-level to Service-level States	38
6	Experimental Validation: Testing Infrastructure	42

Engineering with Logic	3
6.1 Experimental Setup and Instrumentation for Trace Generation	42
6.2 Tests	43
6.3 Distributed Checking Infrastructure	44
6.4 Visualisation Tools and Data Management	44
7 Experimental Validation: Checking Technology	46
7.1 The Core Algorithm: Evaluating One Transition	48
7.2 Laziness in Symbolic Evaluation	49
7.3 Evaluating Time Transitions	50
7.4 Model Translation	50
7.5 Adding Constraints	52
7.6 Simplification	53
7.7 Service-level Validation	53
8 Experimental Validation: Results	55
8.1 Protocol-Level Checking Results	55
8.2 Implementation Anomalies	55
8.3 Service-level Results	56
8.4 Validating the Other Direction	56
9 Revisiting the TCP State Diagram	57
10 Related Work	57
11 Discussion	60
11.1 Using the Specifications	61
11.1.1 As an Informal Reference	61
11.1.2 For Bug-fixing	61
11.1.3 As a Basis for New Implementations	61
11.1.4 For Testing New Implementations	61
11.1.5 As a Reference for Proposed Changes	62
11.1.6 As a New Internet Standard	62
11.1.7 As a Basis for Formal Reasoning	62
11.2 General Lessons for New System and Protocol Development	62
11.2.1 Implementation-internal Nondeterminism	63
11.2.2 Superficial Inter-implementation Variation	63
11.2.3 Debatable Looseness	63
11.2.4 Major Variations	63
References	65

1. INTRODUCTION

We begin by recalling normal industry development practice, to explain the fundamental problem that we address. We highlight some of its difficulties and we focus on the concept of specifications that are executable as test oracles (§1.2) and on the concept of the de facto standards (§1.3), using the TCP/IP network protocols and Sockets API as an example (§1.4). We then introduce our work to develop specifications for TCP, UDP, and the Sockets API (§1.5) and explain two of the main technical challenges involved (§1.6, §1.7). We conclude the introduction by summarising some non-goals and limitations (§1.8), the project history (§1.9), and the structure of the remainder of the paper (§1.10).

1.1. Standard Industry Practice

At present the overwhelming majority of our computing infrastructure is built with a testing-based development process. The only way that we normally have to assess whether implementation code will behave satisfactorily is to execute it on a collection of concrete test cases and examine the outcomes. This leads to the standard test-and-debug development cycle, in which code is written, it is executed on tests, the outcomes are assessed, and the code is rewritten accordingly. For each test, the allowed outcomes are typically defined manually, either as a check within the test itself, or as data on the side.

Computer engineering relies also on the division of labour that is enabled by common interfaces, allowing systems to be composed of parts built by different teams and different organisations. These interfaces include major pan-industry abstractions, such as the processor architectures, programming languages, established libraries, network protocols, filesystem APIs, etc., together with many internal interfaces specific to particular systems. They are typically described with *specification documents* that combine a reasonably precise definition of the interface syntax (variously the processor opcodes, programming language syntax, protocol wire format, API types, etc.) with some informal-prose description of the intended interface behaviour.

Ideally this triumvirate, of code, tests, and specification document, would all be in sync: the defined allowed test outcomes would be correct with respect to the specification-document intent, the tests would have good coverage, both of the code and of the specification document, and the code would pass all the tests. That would provide some confidence (though obviously not certainty) that the code and specification document are consistent, exploiting the redundancy between the three different descriptions to detect errors. One also would hope that a specification document is sufficiently clear and complete to serve as a good description, for both implementers and clients of an interface, of what an implementation must do and what a client can rely on.

However, the normal specification-document reliance on prose to describe behaviour makes both of these ideals hard to attain. Prose has one obvious advantage: specification documents are used to communicate between human beings, and prose makes them superficially accessible. But a prose document cannot be used in that test-and-debug development cycle in any direct, automated way. Instead, one has to manually curate the intended outcomes for each test, defining them and reasoning informally to check that they capture the intent of the prose specification. One also has to manually reason about specification coverage. Prose fails also in that it is intrinsically a poor medium for expressing the subtle and complex behaviour of real systems: with no automated checking or testing of specification documents, their prose descriptions are almost inevitably ambiguous and incomplete, and often in some way inconsistent or simply incomprehensible. It is all too easy to add a paragraph to a prose specification without considering all of its implications. Contrasting prose specification documents with tests, the former cover rich behavioural properties, intended to be readable, subject to ambiguity, and not directly testable; the latter are effectively single-point specifications, not intended to be readable, tolerably precise, but directly testable.

Industry has learned how to get by in this fashion well enough to thrive economically, but the aggregate cost of dealing with the resulting bugs, behavioural oddities, version differences, and security flaws is high. Meanwhile, after 50+ years of research on formal specification and verification,

we have reached the point where a few pieces of nontrivial software that have been designed with verification in mind can be proved correct, but proving functional correctness of substantial bodies of existing “real-world” software is still well out of reach. In fact, even *stating* such correctness properties has been out of reach: for that we would need mathematically rigorous specifications in place of the conventional prose.

1.2. Specifications that are Executable as Test Oracles

Our work develops a middle way between current industry practice and the academic dream of widespread full correctness proofs: we show that it is feasible and useful to build rigorous specifications of a key piece of systems software in such a way that one can test the correspondence between implementation and specification, and one can state and test some important correctness properties, all this despite the many challenges such code embodies.

We do this by focussing on the simple concept of specifications that are *executable as a test oracle*: artifacts that can be used, given a behaviour that might be exhibited by the system, to *compute* whether or not that behaviour is allowed. They need not be decidable in general, so long as they terminate with a yes-or-no answer for the behaviours of sufficient tests. This radically simplifies the construction of test suites: given such a specification, one can make tests by some automatic process, e.g. systematically or randomly enumerating test stimuli, without needing a manually written check or manually curated set of allowed results for each test; instead, one can simply use the test oracle to assess whether the observed behaviour obtained by running each test is allowed.

The concept of a specification that is executable as a test oracle is surely not new, but we believe it to be under-appreciated and under-emphasised, both in industry and in the research literature. It should not be confused with the notion of a specification that is executable in a more conventional sense, that can itself serve as a (perhaps slow) implementation, which can (perhaps pseudorandomly) exhibit any allowed behaviour: a specification that is *executable as a semantically complete implementation*.

A specification that is executable as a test oracle could take many forms. At one extreme, it might be a program in a conventional programming language, that takes a representation of a system behaviour (perhaps a trace) as input, and whose execution checks whether it has some desired properties. At the other, it might be a mathematically rigorous formal specification, equipped with whatever execution mechanism is needed to let one compute whether representations of system behaviour are admitted or not. Much previous work on formal specification has had the primary aim of supporting correctness proofs; for that, mathematical rigour is essential. Supporting testing via executability as a test oracle is a quite different goal; mathematical rigour is desirable for such specifications, to eliminate ambiguity and imprecision within the specification, but it is not in general essential. Sometimes one might have both goals, of course.

Irrespective of this, a specification written to be executable as a test oracle should *also* serve as a readable document, to serve the original purpose of specification documents, of enabling communication and discussion between humans about the intended behaviour. It should be structured to be as clear as possible, which may be in tension with the demands of executability and of rigour, and it should be annotated with explanatory prose and presented in a human form.

1.3. De Facto Standards

Another kind of specification, particularly important for systems in which individual components evolve over time, is that of the *de facto standards*. There are broadly two kinds: the de facto standard implicitly defined by the existing component implementations, as the envelope of all their behaviours, and the de facto standard implicitly defined by the deployed systems that have to interoperate with them, as the conjunction of the assumptions made on their behaviour by all the existing clients. When writing new software above a particular interface, what really matters (in the short term) is the former: the envelope of all behaviour exhibited by the existing implementations that it has to interoperate with. Dually, when writing a new implementation of a particular established interface, what matters is the latter: the assumptions about the interface behaviour that the deployed

clients implicitly rely on; if the new implementation violates one of those assumptions, the deployed clients will fail.

Both of these are hard to investigate with conventional methods, with no way to describe an envelope of allowed behaviour except either (a) the outcomes of all tests from some test suite, or (b) a prose specification document. But a specification that is executable as a test oracle gives a way to experimentally investigate the behaviour of existing implementations, by iteratively generating tests, checking whether the observed behaviour is allowed by the specification, and refining the specification to provide a better approximation. In this way one can develop *post hoc* specifications for existing systems that capture good approximations to the behaviour of the existing component implementations.

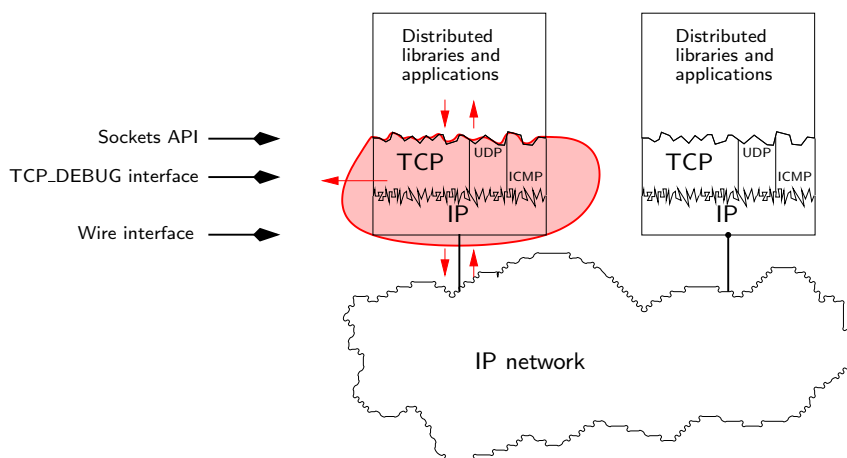
A specification that is executable as a semantically complete implementation would permit the conventional testing of higher-level components above the specification with respect to the full range of allowed behaviour, rather than with respect to the reduced range typical of particular implementations; it would thereby let one investigate the second kind of *de facto* standard: the aggregate of all assumptions about the interface made by clients of the interface.

1.4. The TCP/IP Network Protocols and Sockets API

The TCP/IP network protocols provide a good example of all this, and are the technical focus of our work in this paper. TCP provides a reliable byte-stream communication abstraction between pairs of endpoints that underlies the world-wide web, email, and many other services; the companion UDP protocol provides an unreliable datagram service, often used for streaming media; and both are implemented above IP, the Internet Protocol. This is among the most widely-used software infrastructure on the planet: there are TCP/IP protocol endpoint implementations running on almost all machines, implemented by many different vendors. They should interoperate with each other, but there is no prospect of exhaustively testing all combinations, or of synchronising any updates. By and large the deployed Internet, which is an assembly of all these endpoints and the interconnecting routers, works remarkably well. However, when one looks closely at what the TCP/IP protocols *are*—how they are defined, what the behaviour of those endpoint implementations is (or should be), and in what sense they interoperate correctly—the situation is very unclear.

There are specification documents: RFCs that focus on the on-the-wire protocols, and the POSIX standard for the Sockets API used by applications to interact with the protocols. An RFC is a Request for Comments in a series now published by the Internet Engineering Task Force (IETF); some RFCs are adopted as IETF Internet Standards. RFCs and POSIX are prose documents: they typically 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 protocol and API behaviour. Specifications in this area have to be *loose*, to accommodate implementation variation (e.g. in timing or choice of sequence numbers, window sizes, or, within some bounds, retransmission policy), but, as we shall see, that does not mean that they have to be vague or imprecise.

Then there is the code. For each of the many implementations of TCP/IP and the Sockets API, the code implicitly does define some precise behaviour (modulo the status of the underlying programming language), but there are many differences between them, some intentional and some not. In practice, the common implementations together form a *de facto* standard: any implementation must interoperate reasonably well with all of them, though historically the BSD implementations have a special status: various protocol features were first developed there, and they are sometimes used as a reference [Wright and Stevens 1995]. Moreover, each implementation is in itself a complex body of code. They are typically written in C, intertwined with an operating system. For example, the BSD implementation is around 20 000 lines of C. They are multithreaded, dealing with asynchronous events on the network interface, concurrent Sockets API calls, and the expiry of various timers. At present, there does not exist a formal semantics for the fragment of C that is used, despite much work over the years [Norrish 1998; Papaspyrou 1998; Ellison and Roşu 2012; Krebbers 2015; Memarian et al. 2016; Batty et al. 2011a]), let alone proof techniques that have been shown to scale to this kind of problem. There is a rough layer structure (Sockets/TCP/IP/interface) but much cou-



This shows two TCP endpoints and the intervening IP network; the interface of the protocol-level specification is shown in red, with the shaded region being the part of the real system whose behaviour it models, and the short red arrows indicating its observables.

Fig. 1. Our Single-Endpoint Protocol-level Specification

pling between the layers, with ‘fast path’ optimisations for the common cases, indirection via function pointers, and many historical artifacts. Moreover, the deployed base makes it almost impossible to change many aspects of the observable implementation behaviour, on the wire or Sockets API interface (there is still active experimentation with new congestion control algorithms, however).

Developers, both of protocol stacks and of applications above them, thus have to deal with a very complex world. TCP has long been recognised as hard to implement correctly [Paxson 1997], [RFC2525], and in fact there is no precise sense in which an implementation can be considered ‘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 is clearly possible to write pragmatically satisfactory protocol stacks, and distributed libraries and applications above them, but the cost and level of expertise needed are high.

1.5. Our Specifications

In the light of the above, we set out to investigate whether and how one can do better, taking a mathematically rigorous approach to the behaviour of such real-world systems. We address the specific problem of the unclear de facto standard for TCP/IP, developing specifications of TCP/IP behaviour that are executable as test oracles. This is important in itself, but by doing so — tackling a particularly gnarly real-world abstraction, with all of its difficulties — we can also draw lessons from the experience for the general problem: how one can improve on conventional development processes based on prose specification documents, both to clarify similar legacy abstractions, and in the development of future clean-slate abstractions.

1.5.1. A Protocol-level Specification of the De Facto Standard. We first develop a specification for a TCP or UDP endpoint, as observed in terms of individual TCP and UDP messages on the wire, at its Sockets API interface, and at an existing debug interface. This is illustrated in Fig. 1 (Page 7). We structured this specification and built checking tools to make the specification be executable as a test oracle. This let us develop it with an *experimental semantics* approach, to capture a good approximation of the de facto standard:

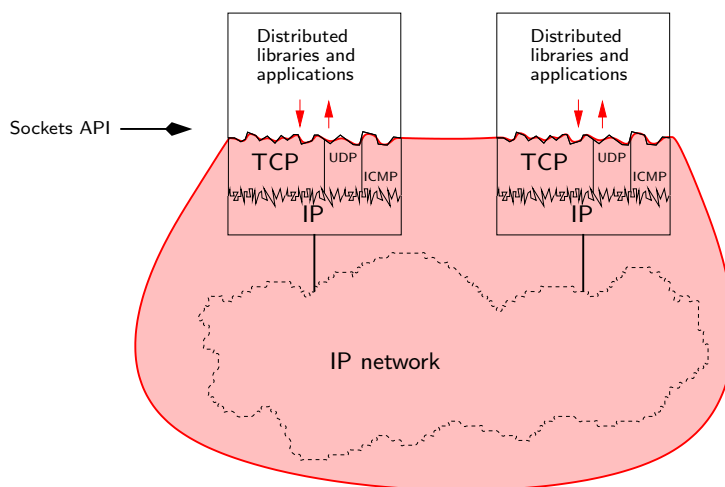
- We produced an initial draft specification based on the existing prose documents, the BSD and Linux source code, and ad-hoc tests.
- In parallel, we instrumented a test network containing machines running multiple implementations, and wrote tests to drive those, generating several thousand real-world traces chosen to cover a wide range of their behaviour. We used three implementations current when the project began: FreeBSD 4.6, Linux 2.4.20-8, and Windows XP SP1. As we discuss later, validation of TCP dynamics focussed on the first of those; for UDP and the Sockets API we used all three.
- We ensured that the specification admitted those traces by running a special-purpose symbolic model checker. When we found that some particular real-world behaviour was not admitted by the specification, we either amended the latter or classified it as an implementation bug. Those bugs, and the many differences between the three implementations that we found, show that this is a very discriminating testing process.
- We iterated this process, improving our tools, understanding, and specification, until the specification captured a good approximation to the observed behaviour.

The resulting specification is fully mathematically rigorous: it is expressed in higher-order logic, in a mechanised proof assistant, HOL4 [Gordon and Melham 1993; HOL 2005], that type-checks the specification. We thereby avoid the ambiguity of prose specifications.

One might be concerned that this rigour comes at the cost of accessibility. Our HOL4 definitions are essentially standard typed discrete mathematics, using numbers, lists, sets, finite maps, logical connectives, quantifiers, and user-defined record types, functions, and relations. This language should be familiar to anyone with previous discrete-mathematics experience. For others (perhaps the majority of developers), and to explain the meaning of the specification more directly, we also *documented* it extensively, with side-by-side English prose and mathematical definitions. Automated typesetting tools let us present the two in a readable form, which also helped us keep them in sync. At a larger scale, we *structured* the specification to be as clear as possible, picking out logically distinct parts of the behaviour into distinct definitions.

HOL4 supports higher-order logic as its definition language. In general, the more sophisticated the language, the more challenging automated reasoning becomes, so one should ask whether this expressiveness was required or whether simpler tools would have sufficed. Higher-order types are used in our specification: there are some third-order types (where we define order by $\text{order}(\text{tycon}) = 0$, $\text{order}(t \rightarrow t') = \text{order}(t \mapsto t') = \max(\text{order}(t) + 1, \text{order}(t'))$, and $\text{order}((t, t', \dots)\text{tyop}) = \max(\text{order}(t), \text{order}(t'), \dots)$). The highest order at which we quantify is order 1, e.g. at the host type.

The resulting document remains large and complex by the standards of most formal models (386pp typeset, 25 800 lines of HOL source, of which two-thirds are comments), but that seems to be inescapable. Most work on formal models either makes significant idealisations, e.g. when defining a “core calculus” for some programming language, or concerns a clean-slate design. In contrast, here we aim to capture TCP as it actually is, without idealisation, in the observable behaviour of some of the deployed implementations. 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, 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 little 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. Our specification is thus *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 routing and other IP internals). And 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



The interface of the service-level specification is shown in red, with the shaded region being the part of the real system whose behaviour it models and the short red arrows indicating its observables.

Fig. 2. Our End-to-end Service-level Specification

— one of our goals was to characterise the failure semantics under network loss, duplication and reordering, API errors, and malicious attack.

Note that this is a *post 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, developing a specification in part by checking it against implementations. However, the checker technology that we develop is symmetric: it could equally well be used to test future implementations against our now-existing protocol-level specification.

Our work also differs from most research on formal models in that our specification is intended principally to support testing and human communication, rather than mechanised proof (except proofs involved in our model checking). In principle the specification should be usable for more general proof, but the scale makes that a challenging prospect, and it was not one of our goals. We have not attempted to prove that any implementation meets the specification, or that the protocol of the specification is correct. We did prove some sanity properties of an early version of the specification.

1.5.2. A Service-level Specification and Validated Abstraction Function. We continue by developing a more abstract *service-level* specification, illustrated in Fig. 2 (Page 9). This abstracts from the details of the individual messages sent on the wire, instead characterising the reliable-stream service that two TCP endpoints combine to provide for applications running above their respective Sockets APIs. This is similarly rigorous, detailed, and with broad coverage. It has much simpler wire behaviour than the protocol-level specification, as one would hope, but still has to expose some aspects of TCP connection setup and teardown, and the complexities of the Sockets API remain largely unchanged.

In principle this gives a meaningful criterion for correctness of the TCP protocol: one would like a proof that two instances of the protocol-level endpoint specification, composed with a model of the network, do implement the service-level specification. But, as mentioned above, the scale of the specifications make that proof a challenging prospect. Instead, we develop a pragmatic testing-based alternative that lets one establish some confidence with much less effort, *experimentally testing an abstraction function*. We defined, in HOL4, an abstraction function that takes a pair of states of the protocol-level model (one for each end of a TCP connection), together with an abstract model of

the state of the intervening IP network (the TCP segments in flight) and constructs a state of the end-to-end service-level model. This function is what one would use as the invariant of a protocol correctness proof. We then built a checker that, given an experimental protocol-level trace for the two endpoints, checks whether there is a corresponding trace of the service-level specification, with the abstraction function mapping protocol-level states onto service-level states at each step.

We now introduce two of the main technical challenges that arise in developing these two specifications: handling specification looseness and relating specification and implementation.

1.6. Technical Challenge: Specification Looseness

A key issue in this work is the need for the specification to be loose, while remaining executable as a test oracle and precise enough to be unambiguous. In a deterministic setting, with a tight specification and no implementation runtime determinism, one can build a specification that is executable as a test oracle by simply writing a reference implementation (perhaps in a conventional programming language, or as a pure functional program), using it as an oracle by checking that the observed behaviour from running tests on a production implementation is identical to the behaviour obtained from running them on the reference implementation. The two notions, of specifications that are executable as a test oracle, and specifications that are executable as a semantically complete implementation, here collapse into one.

In most contexts, however, specification looseness is essential, to accommodate allowed variations in behaviour between different implementations, and to accommodate per-implementation runtime nondeterminism. For TCP, some inter-implementation variation is intended to be allowed by the prose specification documents and some has arisen over time. Then there is a great deal of intra-implementation runtime nondeterminism, arising from OS scheduling, timers, explicitly randomised choices, and so on. Repeatedly running a test case on a single implementation can produce many significantly different traces of observed Sockets API and wire behaviour.

This means that checking conformance cannot be done by simply running the specification and an implementation in lock-step and comparing the results; our specification must be quite different from a “reference implementation” for TCP in a more-or-less conventional programming language, which would exhibit just some behaviours of the many possible (several of these exist, including the BSD C code and those by Biagioni in Standard ML [Biagioni 1994], by Castelluccia *et al.* in Esterel [Castelluccia et al. 1997], and by Kohler *et al.* in Prolac [Kohler et al. 1999]).

Instead, to define an envelope of behaviour that includes such variation, we express our protocol-level specification in a relational style, as a labelled transition system (LTS). This *host LTS* is essentially a nondeterministic automaton, a ternary relation

$$h \xrightarrow{l} h'$$

Here h and h' are values of a HOL4 type modelling a single endpoint state, abstracting from the relevant parts of the OS and network hardware of a machine, and l is a label modelling one of the observable events (shown in Fig. 1 (Page 7)) that the specification deals with: Sockets API calls and returns, network messages sent and received, certain debug trace events, time passage, and internal transitions. The host LTS is defined as an operational-semantics definition in higher-order logic, using various idioms adapted from programming language and concurrency theory semantics, including a relational monad structure and nondeterministic timed labelled transition systems, to structure it as clearly as we can.

Then we have to make this executable as a test oracle. By choosing higher-order logic as the language in which to write the specification, to let us express it as clearly as possible, we already depart from definitions that are immediately executable. But the key difficulty is that much of the nondeterminism, in TCP implementations and in the specification, is *internal*, not immediately exposed in observable events. For example, there may be an internal nondeterministic choice, e.g. of

some TCP sequence number or window size, which only affects the observable behaviour many steps later.

We therefore built a model checker that 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, expressing 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 [Gordon et al. 1979], and so its soundness, and the soundness of our checker above it, depends only on the correctness of a small trusted kernel.

One of the main lessons we learn, that we return to later, is that minimising the amount of internal nondeterminism is highly desirable, at protocol design time and at implementation design time. We aimed originally to support black-box testing of existing unmodified implementations: we did not attempt to add extra instrumentation of their internals, to avoid perturbing the systems we were investigating, and because we planned to test systems for which we did not have source access, e.g. Windows XP. For UDP this was viable, but TCP has much more complex internal behaviour, with internal nondeterminism that is not directly exposed via the Sockets API. The BSD implementation supported a TCP_DEBUG kernel option to expose the protocol state (TCP control block records) at some program points. Including TCP_DEBUG records in our experimentally captured traces let us concretise the symbolic state of the model relatively frequently, resolving constraints and reducing the amount of backtracking. Modern implementations often support more sophisticated tracing frameworks, e.g. DTrace [Cantrill et al. 2004; McKusick et al. 2014], which would simplify this. But ideally, the protocol and API would be designed from the outset to expose all semantically significant internal nondeterministic choices, either with trace events for the specific choices, or with trace events containing the computed corresponding specification state for any implementation state. Given that, the executable-as-test-oracle specification of each transition could be a simple total function, either operating on a fully concrete state and computing a finite description of the possible next transitions (abstracted on input values), or on a triple of a current and next concrete states and transition label, computing whether that transition is allowed. The specification could then be written in any language of total functions, without needing the sophisticated technology of a proof assistant, a symbolic state, or a backtracking search. In short, with careful thought at protocol design-time, much simpler methods than we use here could be used to make a suitable specification that is executable as a test oracle.

On a separate note, there is a common misconception that a rigorous and detailed specification is necessarily also a *tight* specification, overconstraining potential implementations. On the contrary, we have been to great lengths to ensure that our completely-rigorous specifications allow a wide range of reasonable behaviours. We argue that precisely specifying exactly the properties of other protocol endpoints that one depends upon, combined with proof or testing that those properties suffice to ensure good end-to-end system behaviour, would be preferable to an exclusive reliance on the “rough consensus and running code” emphasised in the early development of the internet protocols.

1.7. Technical Challenge: Relating Model and System

The second key issue in making a testable specification of a real system component is that of choosing exactly what part of the real systems the specification is intended to model, and how the two are intended to be related. For many conventional formal specifications, aimed principally at supporting proof about idealised systems, this is not an issue, as they have only a weak relationship to implementations. Some others, such as compiler specifications, have a relatively simple interface, e.g. as partial functions from source to target languages, and hence a relatively straightforward notion of what is observable. In contrast, here we are specifying part of a large and complex system:

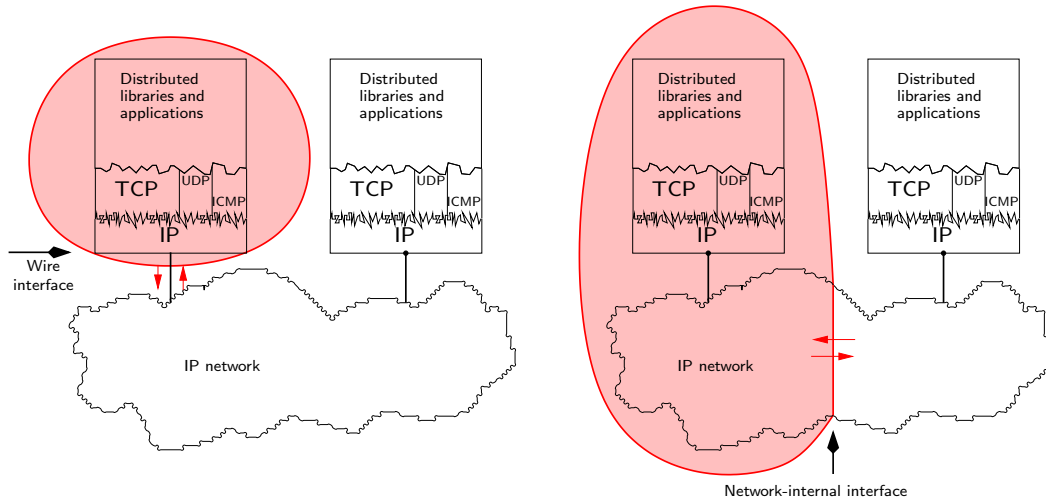


Fig. 3. Wire-interface endpoint and network-internal specifications

the TCP/IP stacks that we consider exist within full operating systems; they are not cleanly isolated components that can be exercised independently (they are also not subject to simple unit testing, for this reason). We therefore have to carefully choose how to restrict the scope to a manageable domain, and what system behaviour is taken as observable. These choices have to be reflected in the top-level form of the judgements of the specification, which ultimately is a predicate on such behaviours, and in the testing infrastructure, as instrumentation that logs the observable behaviour in executions of the running system. For testing-based validation, the two must have some exact correspondence, and this provides an important constraint: we can abstract but cannot idealise. As we are building mathematically rigorous specifications, and the implementations are (as is typical) not mathematical artifacts, this is also the necessarily non-mathematical boundary where we relate the formal and informal worlds.

For TCP/IP there are five main options for where to cut the system to pick out observable events. The *protocol-level* specification of Fig. 1 (Page 7) deals with events at the network interface and Sockets API of a single machine, but abstracts from the implementation details within the network stack. For TCP the obvious wire interface events are at the level of individual TCP segments sent or received, and this level covers the dynamics of how messages are exchanged, so we refer to this as an *endpoint* specification (or sometimes as a *segment-level* specification). The *service-level* specification of Fig. 2 (Page 9) describes the end-to-end behaviour of the network as observed by users of the Sockets API on two different machines, abstracting from what occurs on the wire. For TCP such a specification can model connections roughly as a pair of streams of data, together with additional data capturing the failure behaviours, connection shutdown, etc. We refer to this as a *service-level* specification (or sometimes as a *stream-level* specification), characterising the bidirectional data stream service that the TCP protocol, together with the network and Sockets API, provides to applications. A *wire-interface-only endpoint* specification, shown on the left of Fig. 3 (Page 12), would specify the legal TCP segments sent by a single host irrespective of whatever occurs at the API. A *network interior* specification, shown on the right of Fig. 3 (Page 12), would characterise the possible traffic at a point inside the IP network, of interest for network monitoring. Finally a *pure transport-protocol* specification (not shown) would define the behaviour of just the TCP part of a TCP/IP stack, with events at the Sockets API and at the OS-internal interface to IP.

All would be useful, for different purposes. We chose to begin by developing a protocol-level endpoint specification for three main reasons. Firstly, we considered it essential to capture the be-

haviour at the Sockets API, despite the fact that the usual TCP/IP RFC specifications do not cover the API (it is addressed to some extent in POSIX). Focussing exclusively on the wire protocol would be reasonable if there truly were many APIs in common use, but in practice the Sockets API is also a *de facto* standard, with its behaviour of key interest to a large body of developers. Ambiguities, errors, and implementation differences here are often just as important as for the wire protocol. Secondly, the protocol-level specification has a straightforward model of network failure, essentially with individual segments either being delivered correctly or not; the observable effects of network failure in an end-to-end model are far more easily characterised as a corollary of this rather than directly. Thirdly, it seemed likely that automated validation would be most straightforward for an endpoint model: by observing interactions as close to a host as possible (on the Sockets and wire interfaces) we minimise the amount of nondeterminism in the system and maximise the amount of information our instrumentation can capture. With our protocol-level specification in hand, we were then in a good position to develop a service-level specification, as described in §4.

Then there are more detailed questions about the instrumentation that is feasible and the exact relationship between specification and running system. We return to these later, but to give a few examples:

- A pure transport-protocol specification would require us to instrument the OS-internal interface between TCP and IP, or that between the TCP/IP stack and the network cards. Neither of those are easily accessible. Instead, the protocol-level specification has the advantage that it requires us only to instrument the wire interface, which for ethernet wire interfaces can be done cleanly with another network-interface observer.
- The Sockets API interface is conceptually simple to deal with at first sight, but it is a C language interface, involving exchange of shared memory between application and OS. To avoid having to deal with those intricacies of C semantics in our specification, we abstract the API calls and returns into a pure value-passing model; that abstraction is computed by our instrumentation.
- We have to consider how precisely events in different parts of the system can be timestamped, both to relate to the numerical values of timers in the TCP implementations, and also simply to let them be totally ordered consistently with real time, to avoid unnecessary search in our trace-checking process.

1.8. Non-goals, Non-approaches, and Limitations

The problem we address, and our approach, are rather different to most previous work on program verification and model-checking. Several non-goals and non-approaches of our work have already been mentioned, but for clarity we collect them here. We are not trying to:

- (1) Prove correctness of the TCP protocol. Our protocol-level and service-level specifications let such a result be precisely stated, and we conjecture that such a proof would be possible, but it would be a separate multi-person-year project in mechanised reasoning. Before embarking on it, one would also want to validate the specifications against additional implementations, to ensure they are not overly specific to those we tested against.
- (2) Prove correctness of any particular implementation with respect to our protocol-level specification. The scale and complexity of the specification and the existing C implementations make this an intimidating prospect, though it might be viable for clean-slate implementations in semantically simpler languages.
- (3) Prove correctness of application code above the specification. In principle the specification should support it, but again it would be a major piece of work in itself. Two preliminary experiments have been done: Compton [Compton 2005] verified a simple distributed algorithm, Stenning's protocol, above our early UDP model, and Ridge [Ridge 2009] conducted a verification of a core piece of distributed infrastructure above a simplified TCP model, based on those we describe here.

- (4) Find bugs in TCP implementations. We are treating the existing implementations principally as part of the de facto standard, and from that point of view whatever they do is “correct”. But along the way we did find several clearly unintended behaviours, as we discuss later.
- (5) Redesign TCP. Again, we are taking the existing implementations as our principal reference, not trying to change the protocol they exhibit. We are redesigning TCP in a different sense, of course: the specified protocol and the way it is specified.
- (6) Implement TCP. Our specification is executable as a test oracle, but it is not executable in the conventional sense, to interoperate with existing TCP implementations. It could be interesting and useful to make it executable as a (likely very slow) implementation, by abstracting out the places where it makes nondeterministic choices so that they can be instantiated with particular strategies, and to explore how one can narrow the gap between that and a production-quality implementation.
- (7) Generate tests for TCP from the specifications. Our test cases are generated by a special-purpose handwritten test bench. We did some coverage analysis with respect to the protocol-level specification, but the tests are not generated from the specification.
- (8) Model-check TCP (either specification or implementation). Other model-checking techniques have been applied to substantial bodies of real code, notably BDD, SAT, and SMT-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 functional correctness property which would be hard to state with assertions or simple temporal logics. On the other hand, those methods analyse the source code fairly directly, whereas we consider only its behaviour as manifested in the generated traces.

A TCP endpoint has a very large and complex state space, both in the implementations and in our protocol-level specification. In the latter, the control block for each end of a TCP connection contains 14 32-bit sequence numbers, several natural numbers, 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. 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.

- (9) Distill some essence of TCP, in the form of an idealised model. In our early work in this area we attempted this, defining a “UDP calculus” in the style of existing process calculi, but it quickly became clear that the real challenge is understanding how to cope rigorously with the scale and complexity of the real protocols, APIs, and implementations.

We return to the limitations of our work in the Discussion section (§11) but highlight the main two up-front:

- (1) Further validation would have been desirable. For UDP and the the Sockets API we believe we achieved reasonable coverage for the three implementations we considered, while for the protocol-level TCP specification we seriously addressed only the BSD implementation, and our tests were typically relatively short, not exercising some regimes of the protocol well. We reached a good correspondence between specification and the observed behaviour of implementations — not a perfect one, but good enough to suggest that only modest additional effort would have been needed to fix the remaining issues. For the service-level specification, we were able only to do proof-of-concept validation.

- (2) We did not build a turn-key protocol stack testing tool that could be used in industry, or attempt to engage with the IETF to improve their TCP specifications. Again, these were mainly due to a lack of available resources.

1.9. Project History

This paper gives a retrospective synoptic view of an extended research project: we started around October 2000 with a simple model for UDP and Sockets [Serjantov et al. 2001a; Serjantov et al. 2001b], initially developed by Serjantov (at the start of his PhD) and Sewell. With Wansbrough and Norrish we extended that with modelling of time and multi-threaded Sockets API clients, and mechanised it in HOL [Wansbrough et al. 2002; Norrish et al. 2002]. We then developed the much more substantial protocol-level specification for TCP [Bishop et al. 2005a; Bishop et al. 2006], by Bishop, Fairbairn, Norrish, Sewell, Smith, and Wansbrough, and finally the service-level specification and abstraction function between the two models [Ridge et al. 2008], by Ridge, Norrish, and Sewell. For more details we refer the reader to a technical report [Bishop et al. 2005b] and to the annotated service-level and protocol-level specifications [Bishop et al. 2005b; Bishop et al. 2005c; Ridge et al. 2009]. The whole project was approximately 11 person-years of effort.

The HOL code for both specifications is available on-line, as a snapshot of the original [The Netsem Project] and, updated to a more recent version of HOL4, in a github repository under the simplified BSD license [Netsem github 2015].

We also applied our techniques to a new protocol as part of its design process, in collaboration with the designers: a Media Access Control (MAC) protocol for the SWIFT experimental optically switched network, by Dales and others [Biltcliffe et al. 2006]. We do not detail that here, but do incorporate some of the lessons learned into our discussion.

Other work made use of our UDP and TCP specifications, providing evidence that they can be put to use. Working with our group, Compton [Compton 2005] verified a simple distributed algorithm, Stenning's protocol, above our early UDP model, and Ridge [Ridge 2009] verified a core piece of distributed infrastructure, involving a simplified model of TCP networking based on those we describe, a filesystem, and concurrent OCaml code. Elsewhere, Li and Zdancewic [Li 2008; Li and Zdancewic 2007; Li and Zdancewic] built a purely functional Haskell TCP server implementation closely based on our TCP specification, roughly 3x–8x slower than the Linux kernel TCP stack.

1.10. Paper Structure

We begin with some background about TCP/IP and its existing specifications (§2). We then describe our (low-level) protocol-level specification (§3), (high-level) service-level specification (§4), and the abstraction function between them (§5). Our experimental validation is described in §6, and the symbolic model checking technology we developed for the purpose in §7. The results of this checking are given in §8: both quantitative results and some selected implementation anomalies from among the many we found. We conclude with related work and discussion in §10 and §11.

2. BACKGROUND: TCP/IP, THE SOCKETS API, AND THE EXISTING SPECIFICATIONS

To make this paper as self-contained as possible, we start with a brief overview of the TCP/IP protocols, the Sockets API, and the relevant standards.

2.1. The TCP/IP Protocols

IP (the *Internet Protocol*) allows one machine to send a message (an *IP datagram*) to another. Each machine has one or more *IP addresses*, 32-bit values such as 64.170.98.30 for the IPv4 version of the protocol, or 128-bit values for IPv6. The distributed *domain name system* (DNS) maps textual names, e.g. `www.ietf.org`, to such addresses. IP datagrams have their destination specified as an IP address. They carry a payload of a sequence of bytes and contain also a source address and various additional data. They have a maximum size of 65535 bytes, though many are smaller, constructed

to fit in a 1500 byte Ethernet frame body. IP message delivery is asynchronous and unreliable: IP does not provide acknowledgements that datagrams are received, or retransmit lost datagrams. Message delivery is implemented by a combination of local networks, e.g. ethernet, and of packet forwarding between routers; these may silently drop packets if they become congested. A variety of *routing protocols* are used to establish state in these routers, determining, for any incoming packet, to which neighbouring router or endpoint machine it should be forwarded.

UDP (the *User Datagram Protocol*) is a thin layer above IP that provides multiplexing. It introduces a set $\{1, \dots, 65535\}$ of *ports* at each endpoint; a UDP datagram is an IP datagram with a payload consisting of a source port, a destination port, and a sequence of bytes. Just as for IP, delivery is asynchronous and unreliable.

TCP (the *Transmission Control Protocol*) is a thicker layer above IP that provides bidirectional byte-stream communication. It too uses a set $\{1, \dots, 65535\}$ of ports. A *TCP connection* is typically between an IP address and port of one machine and an IP address and port of another, allowing data (unstructured streams of bytes) to be sent in both directions. The two endpoints exchange *TCP segments* embedded in IP datagrams. The protocol deals with retransmission of lost data, reordering of data that arrives out of sequence, flow control to prevent the end systems being swamped with data faster than they can handle, and congestion control to limit the use of network bandwidth. These all involve much detailed mechanism, which is exposed on the wire interface, that we cannot summarise here.

In addition, ICMP (the *Internet Control Message Protocol*) is another thin layer above IP, primarily used for signalling error conditions to be acted on by IP, UDP, or TCP.

Many other protocols are used for specific purposes, but TCP and UDP above IP are dominant. TCP underlies web (HTTP), file transfer (FTP) and mail protocols, UDP is often used for media streaming, and TCP and UDP together underlie the domain name system. The first widely-available release of these protocols was in 4.2BSD, released in 1983. They are now ubiquitous, with implementations in all the standard operating systems and in many embedded devices.

2.2. The Sockets API

Application code can interact with the TCP/IP protocol implementations via the *sockets interface*, a C language API originating in 4.2BSD with calls `socket()`, `bind()`, `connect()`, `listen()`, etc. The sockets interface is usually used for interaction with UDP and TCP, not for direct interaction with IP. A *socket* itself is a TCP/IP protocol stack datastructure, referred to locally by a file descriptor.

To give a sense of how the Sockets API is used, and how it interacts with the wire protocol, here is a sequence of events that might be involved for a server and a client program to set up a TCP connection between their two machines, and to communicate a string over it. To sidestep the C language intricacies of the API, the calls are given in the notation of our specification, not in C.

- (1) First the server makes three Sockets API calls to establish a listening socket:
 - (a) `socket (SOCK_STREAM)`: to create a TCP socket, returning its file descriptor (say FD 8);
 - (b) `bind (FD 8, ↑ (IP 192 168 0 12), ↑ (PORT 3333))`: to set the local IP address (to one of the IP addresses of the server) and port of that socket; and
 - (c) `listen (FD 8, 3)`: to put the socket into LISTEN state, ready for incoming connections on that IP address and port.
- (2) Then the client makes the following API calls to create a socket and initiate a connection to the server:
 - (a) `socket (SOCK_STREAM)`: to create a TCP socket, returning its file descriptor (this might also be FD 8, though referring to a client socket datastructure rather than the server socket datastructure);
 - (b) `bind (FD 8, *, ↑ (PORT 3333))`: to set the client socket port, leaving its IP address wild-carded; and

- (c) connect (FD 8, IP 192 168 0 14, ↑ (PORT 3333)): specifying the server's IP address and port, to initiate a TCP connection between the two machines.
- (3) The protocol stacks on the two machines then exchange TCP datagrams in a three-message handshake:
 - (a) client sends (and server receives) a TCP *SYN* datagram;
 - (b) server sends (and client receives) a TCP *SYN ACK* datagram; and
 - (c) client sends (and server receives) a TCP *ACK* datagram; and the client connect() call returns.
- (4) After creating its socket, the server (typically in a loop) calls accept (FD 8) to get the next available connection. This is in the form of a freshly created server-side connected socket for this endpoint of the connection; the accept() call returns returns its file descriptor (say FD 9) and the remote IP address and port of the other endpoint, here (IP 192 168 0 14, PORT 3333). The server listening socket remains available for other incoming connections.
- (5) The server calls recv (FD 9, 6, []) to read 6 bytes from its input buffer for the new connection. Initially this will block, as the requested data is not available.
- (6) To send data, the client calls send (FD 8, *, "Hello!", []), to send the string "Hello!" on the connection. This returns as soon as the string is locally queued.
- (7) The client protocol stack sends a TCP datagram containing that string, which is received by the server and added to the input queue of its connected socket.
- (8) The server recv (FD 9, 6, []) call will return with the string.
- (9) ...subsequent actions to close the connection

2.3. Standard Practice: Protocol and API Descriptions

The development process used for the internet protocols was driven by implementation, experiment, and interoperability testing, with informal specifications (in the form of RFCs) aiming to capture enough of this to support further interoperable implementations. This process, summarised in Clark's "rough consensus and running code" [Clark 1992], was in contrast to a more specification-driven approach advocated elsewhere, e.g. by the ISO/ITU-T OSI effort. It is not our intention to revisit that historical argument, or to take a view on what should or could have been done at the time. But the very success of those protocols makes it important to understand the disadvantages and advantages of the standards that we have ended up with. We summarise those here, to set the context for the rest of the paper.

The basic IP, UDP, TCP and ICMP protocols are described in *Request For Comment (RFC)* standards from 1980–81:

User Datagram Protocol	RFC 768	1980	3pp	STD 6
Internet Protocol	RFC 791	1981	iii+45pp	STD 5
Internet Control Message Protocol	RFC 792	1981	21pp	STD 5
Transmission Control Protocol	RFC 793	1981	iii+85pp	STD 7

The sockets interface appears as part of the POSIX standard [IEEE 2000]. Additional information is contained in the documentation for various implementations, in particular the Unix man pages, and well-known texts such as those of Stevens [Stevens 1994; Wright and Stevens 1995; Stevens 1998].

From the titles of these documents the reader might gain the impression that TCP is a single well-defined protocol. Unfortunately that is not the case, for several different reasons.

- As the protocol has been used ever more widely, in network environments that are radically different from that of the initial design, various clarifications and proposals for changes to the protocol have been made. A small sample of later RFCs include:

Requirements for Internet Hosts — Communication Layers	RFC 1122	1989
TCP Extensions for High Performance	RFC 1323	1992
The NewReno Modification to TCP's Fast Recovery Algorithm	RFC 3782	2004

Deployment of these changes is inevitably piecemeal, depending both on the TCP/IP stack implementers and on the deployment of new operating system versions, which —on the scale of the Internet— cannot be synchronised.

- Implementations also diverge from the standards due to misunderstandings, disagreements and bugs. For example, RFC 2525 collects a number of known TCP implementation problems. The BSD implementations have often served as another reference, distinct from the RFCs, for example with the text [Wright and Stevens 1995] based on the 4.4 BSD-Lite code.
- The existence of multiple implementations with differing behaviour gives rise to another ‘standard’: in addition (or as an alternative) to checking that an implementation conforms to the RFCs one can check that it interoperates satisfactorily with the other common implementations. The early RFC791 enshrined the doctrine that implementations should, as far as possible, interoperate even with non-RFC-conformant implementations:

The implementation of a protocol must be robust. Each implementation must expect to interoperate with others created by different individuals. While the goal of this specification is to be explicit about the protocol there is the possibility of differing interpretations. In general, an implementation must be conservative in its sending behavior, and liberal in its receiving behavior. That is, it must be careful to send well-formed datagrams, but must accept any datagram that it can interpret (e.g., not object to technical errors where the meaning is still clear).

This focus on interoperability in the short term opens the door to “standard drift”, with correctness implicitly defined by what other implementations accept rather than any single concrete definition, and perhaps works against interoperability in the long term, as it becomes unclear what one can depend on from other implementations.

- Similarly, existing applications implicitly encode facts about the behaviour of the Sockets API and protocols, and are used as documentation (most programmers will look for examples to illuminate the man pages or to use as a basis for their own code) and as a test suite (a change is unlikely to be allowed into the OS if it breaks a significant number of applications, or a few significant ones). Yet another source is the expert community, including `comp.protocols.tcp-ip`, Stack Overflow, and the Linux kernel mailing list. In practice it can be these that clarify or even identify bugs in the usual sources and in implementations.
- Neither the RFCs nor POSIX attempt to specify the behaviour of the sockets interface in any detail. The RFCs focus on the wire protocols (RFC793 also describes a model API for TCP, but one that bears little resemblance to the sockets interface as it developed); POSIX describes the C types and some superficial behaviour of the API but does not go into details as to how the interface behaviour relates to protocol events.
- Finally, the RFCs and POSIX are informal natural-language documents. Their authors were clearly at pains to be as clear and precise as they could, but almost inevitably the level of rigour is less than that of a mathematical specification, and there are many ambiguities and missing details.

It might be argued that this vagueness is an important positive aspect of the specifications, permitting the specifications to be loose enough to accommodate implementation variation and new

protocol innovation. In an early experimental phase there may be some truth to this, but ultimately it is a very unfortunate confusion: protocol specifications surely do need to be *loose*, for both of those reasons, but that does not imply that they should be vague or imprecise, as we shall see.

3. PROTOCOL-LEVEL SPECIFICATION

We now describe our protocol-level specification. It is written as a mechanized higher-order logic definition in HOL4 [Gordon and Melham 1993; HOL 2005], a language that is rich and expressive yet supports both internal consistency checking (type checking is essential with a definition of this scale) and our automated testing techniques. The specification is large by the standards of formal artifacts: around 9000 non-comment lines of HOL4 source, interspersed with 17000 lines of comment and whitespace; it is typeset automatically into a 386-page document [Bishop et al. 2005c]. Of this, around 125 pages is preamble defining the main types used in the model, e.g. of the representations of host states, TCP segments, etc., and various auxiliary functions. The remainder consists primarily of the host transition rules, each defining the behaviour of the host in a particular situation, divided roughly into the Sockets API rules (160 pages) and the protocol rules (75 pages). This includes extensive comments, e.g. with summaries for each Sockets call, and differences between the model API and the three implementation APIs.

It is obviously impossible to describe all this in detail here. We first give a reasonably complete description of the external interface of the specification (§3.1), to explain the scope of the specification and relationship to actual implementations, continuing the §1.7 discussion. We then describe its internal structure (§3.2) with just a few excerpts: some of the key types of a host state h and three sample transition rules, to give a feel for the specification style and to set the scene for discussion of the specification idioms we had to use (§3.3). We conclude this section with an example TCP trace (§3.4).

Readers familiar with the Sockets API and TCP/IP should find some of the details familiar, and be able to contrast the style of our specification with their experience of RFCs and implementations. Others, perhaps with more experience in logic and semantics, may find some of that networking detail obscure, but should be able to get a sense of what kind and scale of specification is needed here. In contrast with typical papers about the semantics of small calculi, we cannot include or explain all the definitions used in our excerpts. Both groups of readers may therefore want to skim some of the details in §3.1 and §3.2, depending on their preference and background.

By working in a general-purpose proof assistant we have been able to choose specification idioms almost entirely for clarity, not for their algorithmic properties; we return in Sections 6 and 7 to the experimental, algorithmic and proof aspects of checking traces against this specification. We tried hard to establish idioms with as little syntactic noise as possible, e.g. with few explicit ‘frame conditions’ concerning irrelevant quantities.

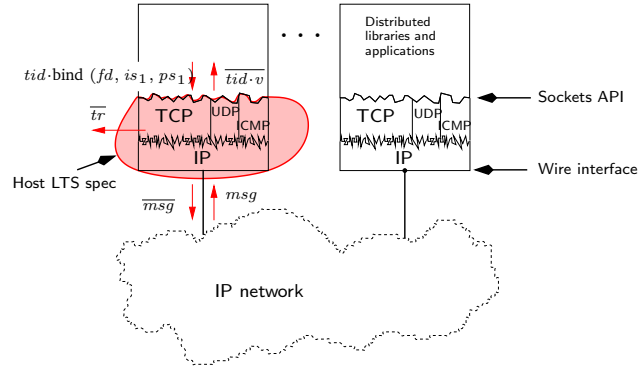
The HOL4 definition language used in the specification is essentially just typed higher-order mathematics. It allows us to build our model quite naturally, using standard data types (pairs, records, sets, lists, lookup tables, numbers, etc.) and defining behaviour declaratively using functions and relations. HOL types can be constructed from type constructors, built-in or user-defined, of natural-number arities. We make extensive use of pair types ($t \# t$), functions ($t \rightarrow t$), finite maps ($t \mapsto t$), labelled records ($\langle \langle fld_1 : t_1, fld_2 : t_2, \dots \rangle \rangle$), 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.

The main HOL expression syntax used is as follows. The notion for logic and sets is standard: \wedge , \vee , \Rightarrow , \forall , \exists , etc. Records are written within angled brackets $\langle \dots \rangle$. Record fields can be accessed by dot notation $h.ifds$ or by pattern-matching. Since all variables are logical, there is no assignment or record update *per se*, but we may construct a new record by copying an existing one and providing new values for specific fields: $cb' = cb \langle \langle irs := seq \rangle \rangle$ states that the record cb' is the same as the

record cb , except that field $cb'.irs$ has the value seq . For optional data items, $*$ denotes absence (or a zero IP or port) and $\uparrow x$ denotes presence of value x . Concrete lists are written $[1, 2, 3]$ and appending two lists is written using an infix $++$. The expression $f \oplus [(x, y)]$ or $f \oplus (x \mapsto y)$ denotes the finite map f updated to map x to y . Finite map lookup is written $f[x]$.

3.1. The External Form and Scope of the Specification

3.1.1. The Top-level Operational Semantics Judgement. The main part of the specification (modelling the shaded region below) is the *host labelled transition system*, or *host LTS*. This describes 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.



Mathematically, the host LTS is simply a ternary 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 a label of one 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;
- $tid \cdot f(arg_1, \dots, arg_n)$ for a Sockets API call f made by thread tid , e.g. $tid \cdot \text{bind}(fd, is_1, ps_1)$ for a $\text{bind}()$ 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 enqueueing other datagrams for output;
- \overline{tr} for a BSD TCP_DEBUG trace record; and
- d for time $d \in \mathbb{R}_{>0}$ passing.

In addition there are labels for loopback messages and changes of network interface status.

The host state type is not technically an abstract type, but here we think of it as such: the labelled transitions are intended to describe all the observable behaviour of the system, to correspond with experimentally observed implementation behaviour as exposed by our instrumentation. We return to the host state type in more detail in §3.2.

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 is done to map them to events in the model. We discuss some of these in the remainder of this subsection, detailing the protocol features we cover and the Sockets API, wire, and debug interfaces of the specification.

3.1.2. Protocol Issues. We restrict our attention to IPv4, though there should be little difference for IPv6. For TCP we cover roughly the protocol developments in FreeBSD 4.6-RELEASE. We

include MSS options; the RFC1323 timestamp and window scaling options; PAWS; the RFC2581 and RFC2582 New Reno congestion control algorithms; and the observable behaviour of syncaches. We do not include the RFC1644 T/TCP (though it is in this codebase), SACK, or ECN. For UDP, for historical reasons (to simplify matters near the start of the project) we deal only with unicast communication.

3.1.3. Network Interface Issues. The network interface events \overline{msg} and msg are the transmission and reception of UDP datagrams, ICMP datagrams, and TCP segments. We abstract from IP, omitting the IP header data except for source and destination addresses, protocol, and payload length. We also abstract from IP fragmentation, leaving our test instrumentation to perform IP reassembly.

Given these abstractions, the model covers unrestricted wire interface behaviour. It describes the effect on a host of arbitrary incoming UDP and ICMP datagrams and TCP segments, not just of the incoming data that could be sent by a ‘well-behaved’ protocol stack. This is important, both because ‘well-behaved’ is not well-defined, and because a good specification should describe host behaviour in response to malicious attack as well as to loss.

Cutting at the wire interface means that our specification models the behaviour of the entire protocol stack together with the network interface hardware. Our abstraction from IP, however, means that only very limited aspects of the lower levels need be dealt with explicitly. For example, a model host has single queues of input and output messages; each queue models the combination of buffering in the protocol stack and in the network interface.

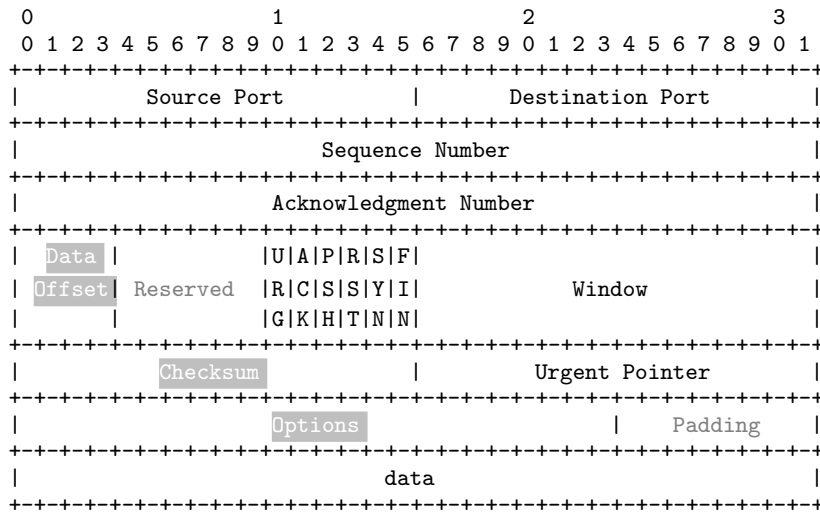
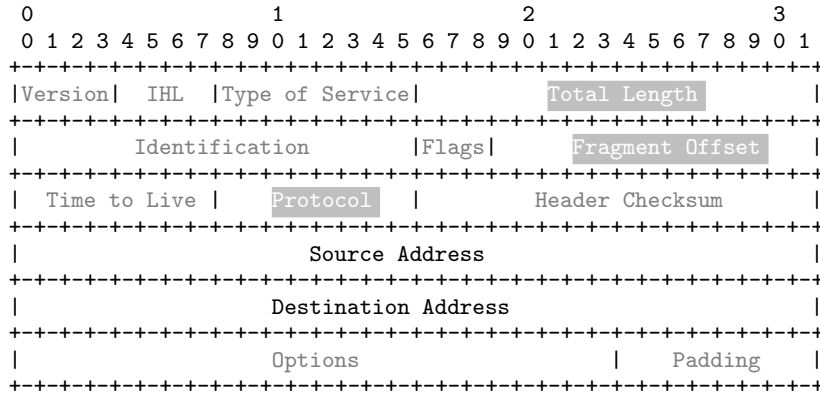
Fig. 4 (Page 22) contrasts the RFC descriptions of IP and TCP headers (from Fig. 4 of RFC 791 and Fig. 3 of RFC 793 respectively) with our specification’s TCP segment type. Fields in gray are not represented in the specification (most are specific to IP), while fields with a gray background are represented in some implicit or partial fashion. Ours is a fairly precise model of a TCP segment: we include all TCP flags and the commonly-used options, but abstract from option order, IP flags, and fragmentation. Address and port values are modelled with HOL option types to allow the zero values to be distinguished by pattern matching; we use the NONE value, typeset $*$, to denote the zero value, and SOME x , typeset $\uparrow x$, for all others.

In retrospect, if doing this again, we would arrange for there to be an exact bijection between values of the model type and the byte sequences comprising real TCP segments, for hygiene. That is not quite true for the type above, but for uninteresting reasons — for example, the type does not record the values of padding, or the order in which TCP options appear. We would moreover express that bijection within the logic, not just in the instrumentation. It might be preferable to support pattern matching against constant values in the prover, to avoid the meaningless $\uparrow 0$ values introduced by the lifting to option types.

3.1.4. Sockets Interface Issues. The Sockets API is used for a variety of protocols. Our model covers only the TCP and UDP usage, for SOCK_STREAM and SOCK_DGRAM sockets respectively. It covers almost anything an application might do with such sockets, including the relevant `ioctl()` and `fcntl()` calls and support for TCP urgent data. Just as for the wire interface, we do not impose any restrictions on sequences of socket calls, though in reality most applications probably use the API only in limited idioms.

The Sockets API is not independent of the rest of the operating system: it is intertwined with the use of file descriptors, IO, threads, processes, and signals. Modelling the full behaviour of all of these would have been prohibitive, so we have had to select a manageable part that nonetheless has broad enough coverage for the model to be useful. The model deals only with a single process, but with multiple threads, so concurrent Sockets API calls are included. We have to split calls and returns into separate transitions to allow interleavings thereof. It deals with file descriptors, file flags, etc., with both blocking and non-blocking calls, and with `pselect()`. The `poll()` call is omitted. Signals are not modelled, except that blocking calls may nondeterministically return EINTR.

The Sockets API is a C language interface, with much use of pointer passing, of moderately complex C structures, of byte-order conversions, and of casts. While it is important to understand these details for programming above the C interface, they are orthogonal to the network behaviour.



```

tcpSegment = {
  is1 : ip option; (* source IP address *)
  is2 : ip option; (* destination IP address *)
  ps1 : port option; (* source port *)
  ps2 : port option; (* destination port *)
  seq : tcp_seq_local; (* sequence number *)
  ack : tcp_seq_foreign; (* acknowledgment number *)
  URG : bool; ACK : bool; PSH : bool; (* flags *)
  RST : bool; SYN : bool; FIN : bool;
  win : word16; (* window size *)
  ws : byte option; (* window scaling option, typically 0..14 *)
  urp : word16; (* urgent pointer *)
  mss : word16 option; (* max segment size option *)
  ts : (ts_seq#ts_seq) option; (* RFC1323 option *)
  data : byte list (* data *)
}

```

Fig. 4. RFC IP and TCP Header Formats vs our TCP specification segment type.

Moreover, a model that is low-level enough to express them would have to explicitly model at least pointers and the application address space, adding much complexity. Accordingly, we abstract from these details altogether, defining a pure value-passing interface. For example, in FreeBSD the `accept()` call has type:

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

In the model, on the other hand, `accept()` has type

```
fd → fd * (ip * port)
```

taking an argument of type `fd` (a model file descriptor) and either returning a triple of type `fd * (ip * port)` or raising one of several possible errors. As part of our instrumentation, the abstraction from the system API to the model API is embodied in an `nssock` C library, which has almost exactly the same behaviour as the standard calls but also calculates the abstract HOL views of each call and return, dumping them to a log.

The model is language-neutral, but we also have an OCaml [Leroy et al. 2004] library implemented above `nssock`, with types almost identical to those of the model, that allows direct programming (the differences are minor, e.g. in using OCaml 31-bit `ints` rather than HOL natural-number `nums` as the `listen-queue-length` argument to `listen()`). This is implemented as a wrapper above `nssock`, and so can also log events. Our library is similar but not identical to that included as part of the `Unix` module with the OCaml distribution.

Fig. 5 (Page 24) gives the complete API that we support (omitting only some mappings to and from the abstract types). Those familiar with the usual Sockets API will be able to see that it is rather complete. We give the OCaml version mentioned above, as it is more concise and easier to read than the HOL version, in which argument and return types of the calls are separated. It uses the subgrammar of OCaml types:

<code>t ::= tc</code>	defined type constructor name
<code>unit</code>	type of the unit value <code>()</code>
<code>bool</code>	booleans
<code>int</code>	integers
<code>string</code>	strings
<code>t1 * t2</code>	pairs <code>(v1, v2)</code> of values of types <code>t1</code> and <code>t2</code>
<code>t1 -> t2</code>	functions from <code>t1</code> to <code>t2</code>
<code>t option</code>	either <code>None</code> or <code>Some v</code> for a value <code>v</code> of type <code>t</code>
<code>t list</code>	lists of values of type <code>t</code>

The type of error codes consists roughly of all the possible Unix errors. Not all those are used in the body of the specification; those that are are described in the ‘Errors’ section of each socket call. The type of signals includes all the signals known to POSIX, Linux, and BSD. The specification does not model signal behaviour in detail (it treats them very nondeterministically), but they occur as an argument to `pselect()` so must be defined here. File descriptors, IP addresses, ports, etc. are abstract types in the OCaml interface, preventing accidental misuse. There are various coercions (which we do not give here) to construct values of these types. For interface identifiers (`ifid`) the specification supposes the existence of a loopback identifier and numbered ethernet identifiers. Any particular host may or may not have an interface with each identifier, of course. The sockets interface involves various flags, for files, sockets, and messages. Both the HOL and OCaml interfaces define them as new types, preventing misuse (though of course this also makes the specification insensitive to such misuse). The OCaml interface indicates error returns to socket calls by an exception, while the HOL4 interface returns error values explicitly. Finally, the figure gives the types of the socket calls themselves. Defining the programmer-visible behaviour of these, in the context of a network and other endpoints, is of course the point of our whole specification.

```

type fd      (* abstract type of file descriptors. In HOL: FD of num *)
type ip      (* abstract type of IP addresses. In HOL: IP of num *)
type port    (* abstract type of inet ports. In HOL: Port of num *)
type netmask (* abstract type of netmasks. In HOL: NETMASK of num *)
type ifid    (* abstract type of interface IDs. In HOL: LO | ETH of num *)

(* error codes *)
type error =
| E2BIG
| EACCES
| ...

(* signals *)
type signal =
| SIGABRT
| SIGALRM
| ...

(* boolean file flags *)
type filebflag =
| O_NONBLOCK
| O_ASYNC

(* boolean socket flags *)
type sockbflag =
| SO_BSDCOMPAT
| SO_REUSEADDR
| SO_KEEPALIVE
| SO_OOINLINE
| SO_DONTROUTE

(* numeric socket flags *)
type socknflag =
| SO_SNDBUF
| SO_RCVBUF
| SO_SNDLOWAT
| SO_RCVLOWAT

(* time socket flags *)
type socktflag =
| SO_LINGER
| SO_SNDTIMEO
| SO_RCVTIMEO

(* boolean message flags *)
type msgbflag =
| MSG_PEEK
| MSG_OOB
| MSG_WAITALL
| MSG_DONTWAIT

(* UDP and TCP socket types *)
type sock_type =
| SOCK_DGRAM
| SOCK_STREAM

(* Sockets API calls *)
exception Unix_error of error * string * string (* in HOL: return OK v or FAIL err *)
val accept: fd -> fd * (ip * port)
val bind: fd -> ip option -> port option -> unit
val close: fd -> unit
val connect: fd -> ip -> port option -> unit
val disconnect: fd -> unit
val dup: fd -> fd
val dupfd: fd -> int -> fd
val getfileflags: fd -> filebflag list
val setfileflags: fd -> filebflag list -> unit
val getifaddrs: unit -> (ifid * ip * ip list * netmask) list
val getsockname: fd -> ip option * port option
val getpeername: fd -> ip * port
val getsockbopt: fd -> sockbflag -> bool
val getsocknopt: fd -> socknflag -> int
val getsocktopt: fd -> socktflag -> (int * int) option
val getsockerr: fd -> unit
val getsocklistening: fd -> bool
val listen: fd -> int -> unit
val pselect: fd list -> fd list -> fd list -> (int * int) option -> signal list option
-> fd list * (fd list * fd list)
val recv: fd -> int -> msgbflag list -> (string*((ip option*port option)*bool) option)
val send: fd -> (ip * port) option -> string -> msgbflag list -> string
val setsockbopt: fd -> sockbflag -> bool -> unit
val setsocknopt: fd -> socknflag -> int -> unit
val setsocktopt: fd -> socktflag -> (int * int) option -> unit
val shutdown: fd -> bool -> bool -> unit
val socketatmark: fd -> bool
val socket: sock_type -> fd

```

Fig. 5. The Sockets API of the specification (OCaml version)

3.1.5. *The TCP Control Block and TCP_DEBUG Interface Issues.* A key part of the BSD TCP implementation, probably similar in most other implementations, is the *TCP Control Block* (TCBCP): a structure containing many of the quantities associated with an endpoint of a TCP connection, including its timers, sequence numbers, reassembly segment queue, etc. Many of these are quantities used in the RFCs to describe the intended dynamic behaviour of the protocol. The BSD implementation we used supported a TCP_DEBUG kernel option to record snapshots of this data to a ring buffer and let it be read out.

In our specification, connections have a similar structure, though with pure values rather than C data structures: a 44-field HOL record type (not shown). We exploit the TCP_DEBUG option to add trace records giving the values of these fields whenever possible. During trace checking, this lets us ground the symbolic state of the model earlier than would otherwise be possible, which is good for efficiency and for early and discriminating detection of mismatches between model and experimental trace.

This is complicated by the fact that TCP_DEBUG adds trace records at somewhat arbitrary program points in the code: there are not enough to fully ground the state, and at some points some of the fields of the record are not meaningful. As mentioned in §1.6, with due design-time care and modest implementation effort, one could have the protocol designed-in instrumentation calculate the whole abstract host state at each interesting point.

3.2. The Internal Structure of the Specification

3.2.1. *Internal Structure: Statics (Host States and Types).* Host states h are simply values of a certain carefully-designed HOL record type:

```

host = ⟨⟨ arch : arch;                (* OS version *)
        privs : bool;              (* whether process has privilege *)
        ifds : ifid ↦ ifd;         (* network interfaces *)
        rttab : routing_table;     (* routing table *)
        ts : tid ↦ hostThreadState timed; (* host view of each thread state *)
        files : fid ↦ file;        (* open file descriptions *)
        socks : sid ↦ socket;     (* sockets *)
        listen : sid list;        (* list of listening sockets *)
        bound : sid list;         (* bound sockets in order *)
        iq : msg list timed;      (* input queue *)
        oq : msg list timed;      (* output queue *)
        bndlm : bandlim_state;    (* bandlimiting *)
        ticks : ticker;          (* kernel timer *)
        fds : fd ↦ fid            (* process file descriptors *)
        ⟩⟩

```

These fields abstract to differing extents from the C representations of actual implementations. Some are quite abstract, with just enough information for the rest of the semantics. For example:

- *ifds* is a finite map from a type *ifid* of network interface identifiers to a type *ifd* of interface descriptors, which just contain the IP addresses, netmask, and up/down status of the interface.
- *privs*, *files*, and *fds* are a minimal representation of the host process's privileges and file descriptor state.

This information was added during specification development as needed, when it became clear that some aspect of the protocol or Sockets API behaviour depended on it.

The *ts* field ties the host semantics to that of the threads making Sockets API calls on the host. It holds a lookup table from thread IDs *tid* to the OS view of thread states (running, blocked in a system call, or ready to be returned a value).

The low-level network interface is modelled with input *iq* and output queues *oq* that are just lists of IP messages (either either a *tcpSegment*, as in Fig. 4 (Page 22), an *icmpDatagram*, or a

udpDatagram). These lists abstract from all the messages queued in the OS or network interface hardware. These, and various other components, are annotated with time information: for any type t , t timed is a type of t values together with elapsed duration and min and max time values. For example, the iq timer is used to model the delay between receipt of the receipt of IP datagrams and their processing, and the thread state timers are used to model scheduling delays. There is also state (bandlim_state) for per-host bandwidth limiting, which applies across all its connections.

The information about sockets (the endpoints and potential endpoints of TCP connections and of UDP communications) have to be represented in more detail, as this is intimately involved in the protocol behaviour. The $socks$, $listen$, and $bound$ fields use the type socket, shown below; this and the types of its components are closer to (a pure value representation of) the C types found in a typical OS implementation.

The $socks$ field of a host h contains a finite map from socket identifiers to sockets. Each socket has internal structure as below, some of which is protocol-dependent: e.g., flags, local and remote IP addresses and ports, pending error, TCP state, send and receive queues, and TCP control block (cb) variables.

```
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 *)
        )
```

For a TCP sockets, the protocol-specific information is as below.

```
tcp_socket = ( st : tcpstate;        (*LISTEN, ESTABLISHED, TIME_WAIT, etc.*)
              cb : tcpcb;           (*the TCP control block*)
              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*)
            )
```

Here the TCP control block, cb (not shown), is the record of sequence numbers, timers, etc. mentioned in §3.1.5. Much of the dynamics of the TCP protocol is specified using these, and many have a close relationship to quantities in the RFC specifications.

3.2.2. Internal Structure: Dynamics (Host Transition Rules). The host LTS is defined in an operational semantics style as the least relation $h \xrightarrow{l} h'$ 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.

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 information hiding, parallel composition

structure, or synchronisation within a host; each rule can refer to any part of the host state as needed. Each transition rule is abstractly of the form

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

where P is a condition (on the free variables of h , l , and h') under which host state h can make a transition labelled l to host state h' . The condition is usually written below the transition. Each rule has a name, e.g. *bind_5*, *deliver_in_1* etc., the protocol for which they are relevant (TCP, UDP, or both), and various categories (e.g. *FAST* or *BLOCK*).

We now give three sample rules, from the 194 that express the dynamic behaviour of the specification. *bind_5* is one of the simplest Sockets API rules, to let the reader get started, and also illustrates a failure case; *send_3* is a more typical non-failure Sockets API rule that shows how we deal with blocking calls and with data manipulation; and *deliver_in_1* is a wire-interface rule, of intermediate complexity, showing how an incoming segment is processed. In the next subsection (§3.3) we discuss some of the specification idioms used in these and other rules.

3.2.3. Sample Protocol-level Transition Rule: *bind_5*. The *bind_5* rule, one of the simplest, is shown in Fig. 6 (Page 28). This is one of 7 rules for the Sockets API call `bind()`. It deals with the case where a thread *tid* calls `bind(fd, is1, ps1)` to set the IP address and port of a socket referenced by the file descriptor *fd* that already has its local port bound; the error `EINVAL` will be returned to the thread.

The **Description** and **Variations** are documentation annotation, rather than part of the rule definition; the former gives an overview of the rule, detailed enough for informal use and checked (manually) against the mathematics, while the latter highlights any differences between implementations that the rule captures.

The rule itself consists of a transition, from one host state via a transition with a `bind` label to another, subject to the sidecondition below. In the host state before the transition, on the first line, the thread state map *ts* maps thread id *tid* to $(\text{RUN})_d$, indicating that the thread is running (in particular, it is not currently engaged in a socket call). In the host state after the transition, on the third line, that thread is mapped to $(\text{RET}(\text{FAIL EINVALID}))_{\text{sched_timer}}$, indicating that within time `sched_timer` the failure `EINVALID` should be returned to the thread (all returns are handled by a single rule *return_1*, which generates labels $\overline{tid.v}$).

The sidecondition is a conjunction of 5 clauses. The first three ensure (line 1) that the file descriptor *fd* is in the host's file descriptor map *h.fds*, (line 2) that *fid* is the file identifier for this file descriptor, and (line 3) that this *fid* is mapped by the host's files map *h.files* to `FILE(FT_SOCKET(sid), ff)`, i.e. to a socket identifier *sid* and file flags *ff*. The fourth (line 4) simply picks out the socket structure *sock* associated with the socket id *sid*. The fifth (lines 5–8) says that the local port 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.

3.2.4. Sample Protocol-level Transition Rule: *send_3*. A slightly more complex rule is *send_3*, in Fig. 7 (Page 29). This rule describes a host with a blocked thread attempting to send data to a socket. The thread becomes unblocked and transfers the data to the socket's send queue. The send call then returns to the user.

As before, the transition appears at the top:

$$h \langle \dots \rangle \xrightarrow{\tau} h \langle \dots \rangle$$

where the thread pointed to by *tid* and the socket pointed to by *sid* are unpacked from the original and final hosts, along with the send queue *sndq* for the socket. Host fields that are modified in the transition are highlighted. The initial host has thread *tid* in state `SEND2`, blocking attempting to send *str* to *sndq*. After the transition, *tid* is in state `RET(OK...)`, about to return to the user with *str''*, the data that has not been sent, here constrained to be the empty string. In contrast to *bind_5*,

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

$$\frac{h \langle ts := ts \oplus (tid \mapsto (\text{RUN})_d) \rangle \quad tid \cdot \text{bind}(fd, is_1, ps_1)}{h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{FAIL EINTR}))_{\text{sched_timer}}) \rangle}$$

1. $fd \in \text{dom}(h.fds) \wedge$
2. $fd = h.fds[fd] \wedge$
3. $h.files[fd] = \text{FILE}(\text{FT_SOCKET}(sid), ff) \wedge$
4. $h.socks[sid] = sock \wedge$
5. $(sock.ps_1 \neq * \vee$
6. $(\text{bsd_arch } h.arch \wedge sock.pr = \text{TCP_PROTO}(tcp_sock) \wedge$
7. $(sock.cantsndmore \vee$
8. $tcp_sock.cb.bsd_cantconnect)))$

Description From thread tid , which is in the RUN state, a $\text{bind}(fd, is_1, ps_1)$ call is made where fd refers to a socket $sock$. The socket already has a local port binding: $sock.ps_1 \neq *$, and rebinding is not supported.

A $tid \cdot \text{bind}(fd, is_1, ps_1)$ transition is made, leaving the thread state $\text{RET}(\text{FAIL EINTR})$.

Variations

FreeBSD	This rule also applies if fd refers to a TCP socket which is either shut down for writing or has its $bsd_cantconnect$ flag set.
---------	---

Fig. 6. Sample protocol-level specification transition rule: $bind_5$

$send_3$ **tcp: slow nonurgent succeed** Successfully return from blocked state having sent data

$$h \langle ts := ts \oplus (tid \mapsto (\text{SEND2}(sid, *, str, opts))_d);$$

$$socks := socks \oplus [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$

$$\text{TCP_Sock}(st, cb, *, \text{sndq}, \text{sndurp}, rcvq, rcvurp, iobc)))] \rangle$$

$$\xrightarrow{\tau}$$

$$h \langle ts := ts \oplus (tid \mapsto (\text{RET}(\text{OK}(\text{implode } str''))_{\text{sched_timer}}));$$

$$socks := socks \oplus [(sid, \text{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore,$$

$$\text{TCP_Sock}(st, cb, *, \text{sndq} ++ str', \text{sndurp}', rcvq, rcvurp, iobc)))] \rangle$$

1. $st \in \{\text{ESTABLISHED}; \text{CLOSE_WAIT}\} \wedge$
2. $space \in \text{send_queue_space}(sf.n(\text{SO_SNDBUF}))$
3. $(\text{length } \text{sndq})(\text{MSG_OOB} \in opts)$
4. $h.\text{arch } cb.t_maxseg \ i_2 \wedge$
5. $space \geq \text{length } str \wedge$
6. $str' = str \wedge str'' = [] \wedge$
7. $\text{sndurp}' = \text{if } \text{MSG_OOB} \in opts \text{ then } \uparrow(\text{length}(\text{sndq} ++ str') - 1) \text{ else } \text{sndurp}$

Description Thread tid is blocked in state $\text{SEND2}(sid, *, str, opts)$ where the TCP socket sid has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state ESTABLISHED or CLOSE_WAIT .

The space in the socket's send queue, $space$ (calculated using send_queue_space), is greater than or equal to the length of the data to be sent, str . The data is appended to the socket's send queue and the call successfully returns the empty string. A τ transition is made, leaving the thread state $\text{RET}(\text{OK}(""))$. If the data was marked as out-of-band, $\text{MSG_OOB} \in opts$, then the socket's urgent pointer will be updated to point to the end of the socket's send queue.

Model details The data to be sent is of type string in the $\text{send}()$ call but is a byte list when the datagram is constructed. Here the data, str is of type byte list and in the transition **implode** str is used to convert it into a string.

Fig. 7. Sample protocol-level specification transition rule: $send_3$

this rule is a purely internal transition of the model, with a τ label; it does not involve any Sockets API or wire events.

The bulk of the rule is the condition (a predicate) guarding the transition, specifying when the rule applies and what relationship holds between the input and output states. The condition is simply a conjunction of clauses, with no temporal ordering. The rule only applies if (line 1) the state of the socket, st , is either ESTABLISHED or CLOSE_WAIT. Then, (lines 2–5) provided $send_queue_space$ is large enough, (line 6 and the final host state) str is appended to the $sndq$ in the final host. Lastly, (line 7) the urgent pointer $sndurp'$ is set appropriately.

3.2.5. Sample Protocol-level Transition Rule: deliver_in_1. A more interesting rule, *deliver_in_1*, for connection establishment, is shown in Fig. 8 (Page 31) (eliding a few details of the TCP protocol TIME_WAIT handling for space). This rule models the behaviour of the system on processing an incoming TCP SYN datagram addressed to a listening socket. It is of intermediate complexity: many rules are rather simpler than this, a few more substantial.

The transition $h \langle \dots \rangle \xrightarrow{\tau} h \langle \dots \rangle$ appears at the top: the input and output queues are unpacked from the original and final hosts, along with the listening socket pointed to by sid and the newly-created socket pointed to by sid' .

As before, the bulk of the rule, below the line, is the condition (a predicate) guarding the transition, specifying when the rule applies and what relationship holds between the input and output states. The condition is simply a conjunction of clauses, with no temporal ordering.

Notice first that the rule applies only when dequeuing of the topmost message on the input queue iq (as defined by predicate `dequeue_iq`) results in a TCP segment seg , leaving remaining input queue iq' . The rule then (block 1) unpacks and constrains the fields of seg by pattern matching: $seg.is_1$ must be nonzero (hence \uparrow) and is bound to variable i_2 ; similarly for i_1, p_2, p_1 ; fields $seg.seq$ and $seg.ack$ are bound to seq and ack (cast to the type of foreign and local sequence number respectively); field $seg.URG$ is ignored (along with FIN and PSH), and so we existentially bind it; of the other TCP flags, ACK is false, RST is false, SYN is true; and so on.

After (blocks 2–4) some validity checks, and determining the matching socket, the predicate computes values required to generate the response segment and to update the host state. For instance, (block 9) the host nondeterministically may or may not wish to do timestamping (here the nondeterminism models the unknown setting of a configuration parameter). Timestamping will be performed if the incoming segment also contains a timestamping request. Several other local values are specified nondeterministically: (block 8) the advertised MSS may be anywhere between 1 and 65495, (block 11) the initial sequence number is chosen arbitrarily, (block 12) the initial window is anywhere between 0 and the maximum allowed bounded by the size of the receive buffer, and so on. Buffer sizes are computed (block 10) based on the (nondeterministic) local and (received) remote MSS, the existing buffer sizes, whether the connection is within the local subnet or not, and the TCP options in use. The algorithm used differs between implementations, and is specified in the auxiliary function `calculate_buf_sizes` (definition not shown).

Finally, (block 16) the internal TCP control block cb' for the new socket is created, based on the listening socket's cb . Timers are restarted, sequence numbers are stored, TCP's sliding window and congestion window are initialised, negotiated connection parameters are saved, and timestamping information is logged. An auxiliary function `make_syn_ack_segment` constructs (block 17–18) an appropriate response segment using parameters stored in cb' ; if the resulting segment cannot be queued (due to an interface buffer being full or for some other reason) then certain of the updates to cb' are rolled back.

Some non-common-case behaviour is visible in this rule: (1) in the BSD implementation it is possible for a listening socket to have a peer address specified, and we permit this (block 4) when checking the socket is correctly formed; and (2) (block 1 pattern) URG or FIN may be set on an initial SYN, though this is ignored by all implementations we consider.

<p><i>deliver_in_1</i> tcp: network nonurgent Passive open: receive SYN, send SYN,ACK</p> <p>$h \{ \text{socks} := \text{socks} \oplus [(sid, \text{sock})]; (* \text{ listening socket} *)$ $iq := iq; (* \text{ input queue} *)$ $oq := oq; (* \text{ output queue} *)$</p> <p>$\xrightarrow{\tau}$</p> <p>$h \{ \text{socks} := \text{socks} \oplus$ $(* \text{ listening socket} *)$ $[(sid, \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, csm, crm,$ $\text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis', [], *, [], *, \text{NO_OOB})));$ $(* \text{ new connecting socket} *)$ $(sid', \text{SOCK}(*, sf', \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, csm, crm,$ $\text{TCP_Sock}(\text{SYN_RCVD}, cb'', *, [], [], *, \text{NO_OOB})));$ $iq := iq';$ $oq := oq' \}$</p> <hr/> <p>(*1. check first segment matches desired pattern; unpack fields *) $\text{dequeue_iq}(iq, iq', \uparrow(\text{TCP_seg})) \wedge$ $(\exists win_ws_mss_PSH \text{ URG } FIN \text{ wrp } data \text{ ack}.$ $seq =$ $\{ is_1 := \uparrow i_2; is_2 := \uparrow i_1; ps_1 := \uparrow p_2; ps_2 := \uparrow p_1;$ $seq := \text{tcp_seq_flip_sense}(seq : \text{tcp_seq_foreign});$ $ack := \text{tcp_seq_flip_sense}(ack : \text{tcp_seq_local});$ $URG := \text{URG}; ACK := \text{F}; PSH := \text{PSH};$ $RST := \text{F}; SYN := \text{T}; FIN := \text{FIN};$ $win := win_; ws := ws_; wrp := wrp; mss := mss_; ts := ts;$ $data := data$ $\} \wedge$ w2n $win_ = win \wedge (* \text{ type-cast from word to integer} *)$ option_map ord $ws_ = ws \wedge$ option_map w2n $mss_ = mss \wedge$</p> <p>(*2. IP addresses are valid for one of our interfaces *) $i_1 \in \text{local_ips } h.\text{ifds} \wedge$ $\neg(\text{is_broadormulticast } h.\text{ifds } i_1) \wedge \neg(\text{is_broadormulticast } h.\text{ifds } i_2) \wedge$</p> <p>(*3. sockets distinct; segment matches this socket; unpack fields of socket *) $sid \notin (\text{dom}(\text{socks})) \wedge sid' \notin (\text{dom}(\text{socks})) \wedge sid \neq sid' \wedge$ $\text{tcp_socket_best_match } \text{socks}(sid, \text{sock}) \text{seg } h.\text{arch} \wedge$ $\text{sock} = \text{SOCK}(\uparrow fid, sf, is_1, \uparrow p_1, is_2, ps_2, es, csm, crm,$ $\text{TCP_Sock}(\text{LISTEN}, cb, \uparrow lis, [], *, [], *, \text{NO_OOB})) \wedge$</p> <p>(*4. socket is correctly specified (note BSD listen bug) *) $((is_2 = * \wedge ps_2 = *) \vee$ $(\text{bsd_arch } h.\text{arch} \wedge is_2 = \uparrow i_2 \wedge ps_2 = \uparrow p_2)) \wedge$ $(\text{case } is_1 \text{ of } \uparrow i1' \rightarrow i1' = i_1 \parallel * \rightarrow \text{T}) \wedge$ $\neg(i_1 = i_2 \wedge p_1 = p_2) \wedge$</p> <p>(*5. (elided: special handling for TIME_WAIT state, 10 lines) *)</p> <p>(*6. place new socket on listen queue *) $\text{accept_incoming_q0 } lis \text{ T} \wedge$</p> <p>(*7. (elided: if drop_from_q0, drop a random socket yielding q0') *) $lis' = lis \{ q_0 := sid' :: q_0' \} \wedge$</p> <p>(*8. choose MSS and whether to advertise it or not *) $advms \in \{ n \mid n \geq 1 \wedge n \leq (65535 - 40) \} \wedge$ $advms' \in \{ *, \uparrow advms \} \wedge$</p> <p>(*9. choose whether this host wants timestamping; resolve with peer *) $tf_rcvd_tstamp' = \text{is_some } ts \wedge$ $(\text{choose } \text{want_tstamp} :: \{ \text{F}; \text{T} \},$ $tf_doing_tstamp' = (tf_rcvd_tstamp' \wedge \text{want_tstamp})) \wedge$</p>	<p>(*10. calculate buffer size and related parameters *) $(rcvbufsize', sndbufsize', t_maxseg', snd_cwnd') =$ $\text{calculate_buf_sizes } advms \text{ mss} * (\text{is_localnet } h.\text{ifds } i_2)$ $(sf.n(\text{SO_RCVBUF}))(sf.n(\text{SO_SNDBUF}))$ $tf_doing_tstamp' h.\text{arch} \wedge$ $sf' = sf \{ n := \text{finupd_list } sf.n(\text{SO_RCVBUF}, rcvbufsize');$ $(\text{SO_SNDBUF}, sndbufsize') \} \} \wedge$</p> <p>(*11. choose whether this host wants window scaling; resolve with peer *) $req_ws \in \{ \text{F}; \text{T} \} \wedge$ $tf_doing_ws' = (req_ws \wedge \text{is_some } ws) \wedge$ $(\text{if } tf_doing_ws' \text{ then}$ $rcv_scale' \in \{ n \mid n \geq 0 \wedge n \leq \text{TCP_MAXWINSIZE} \} \wedge$ $snd_scale' = \text{option_case } 0 \text{ I } ws$ else $rcv_scale' = 0 \wedge snd_scale' = 0) \wedge$</p> <p>(*12. choose initial window *) $rcv_window \in \{ n \mid n \geq 0 \wedge$ $n \leq \text{TCP_MAXWIN} \wedge$ $n \leq sf.n(\text{SO_RCVBUF}) \} \wedge$</p> <p>(*13. record that this segment is being timed *) $(\text{let } t_rttseg' = \uparrow(\text{ticks_of } h.\text{ticks}, cb.\text{snd_nxt}) \text{ in}$</p> <p>(*14. choose initial sequence number *) $iss \in \{ n \mid \text{T} \} \wedge$</p> <p>(*15. acknowledge the incoming SYN *) $\text{let } ack' = seq + 1 \text{ in}$</p> <p>(*16. update TCP control block parameters *) $cb' =$ $cb \{ tt_keep := \uparrow(((\text{slow_timer } \text{TCPTV_KEEP_IDLE});$ $tt_rext := \text{start_tt_rextint } h.\text{arch } 0 \text{ F } cb.t_rttinf;$ $iss := iss; irs := seq;$ $rcv_wnd := rcv_window; tf_rxwin0sent := (rcv_window = 0);$ $rcv_adv := ack' + rcv_window; rcv_nxt := ack';$ $snd_una := iss; snd_max := iss + 1; snd_nxt := iss + 1;$ $snd_cwnd := snd_cwnd'; rcv_up := seq + 1;$ $t_maxseg := t_maxseg'; t_advms := advms';$ $rcv_scale := rcv_scale'; snd_scale := snd_scale';$ $tf_doing_ws := tf_doing_ws';$ $ts_recent := \text{case } ts \text{ of}$ $* \rightarrow cb.ts_recent \parallel$ $\uparrow(ts_val, ts_ecr) \rightarrow (ts_val)_{\text{TIMEWINDOW}} \text{kern_timer } dt\text{sinval};$ $\text{last_ack_sent} := ack';$ $t_rttseg := t_rttseg';$ $tf_req_tstamp := tf_doing_tstamp';$ $tf_doing_tstamp := tf_doing_tstamp'$ $\} \} \wedge$</p> <p>(*17. generate outgoing segment *) $\text{choose } seg' :: \text{make_syn_ack_segment } cb'$ $(i_1, i_2, p_1, p_2)(\text{ticks_of } h.\text{ticks}).$</p> <p>(*18. attempt to enqueue segment; roll back specified fields on failure *) $\text{enqueue_or_fail } \text{T } h.\text{arch } h.\text{rttab } h.\text{ifds}[\text{TCP } seg'] oq$ $(cb$ $\{ snd_nxt := iss;$ $snd_max := iss;$ $t_maxseg := t_maxseg';$ $\text{last_ack_sent} := \text{tcp_seq_foreign } 0w;$ $rcv_adv := \text{tcp_seq_foreign } 0w$ $\}) cb'(cb'', oq')$</p>
---	--

Fig. 8. Sample protocol-level specification transition rule: *deliver_in_1*

3.3. Specification Idioms

Having seen those examples, we now discuss more generally the specification idioms that we used, to capture the range of behaviour required while maintaining a readable specification.

3.3.1. 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 etc. that we saw in blocks 8, 11, and 12 of Fig. 8 (Page 31), the variations due to varying OS scheduling of multiple threads and interrupts, and variations in the rates of timers.

This nondeterminism leads 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).

As discussed in §1.6, if runtime nondeterministic choices were promptly announced in some form, we could instead have phrased the specification as a pure function that takes an observed (non- τ) transition and determines whether or not it is allowed. That might be easier to work with as an executable test oracle, but it would be oriented specifically towards checking — and in any case, the current implementations and RFC/POSIX specifications do not announce those choices.

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* in Fig. 6 (Page 28), 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.

3.3.2. 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

$$t \rightarrow t\#t'\#\text{bool} \rightarrow \text{bool}$$

where t is the state being 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 `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 `andThen` is associative, and so forth, though we have not checked this within HOL.

For *deliver_in_3* we expose four intermediate states: after initial processing (PAWS etc.), after processing the segment’s acknowledgement number and congestion control, after extraction of reassembled data, and after possibly changing the socket’s TCP connection state. The full rule, describing the state changes between each of these points, is around 17 heavily commented pages.

3.3.3. Time and Urgency. It is essential to model time passage explicitly: 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 `dschedmax` (the `sched_timer` of the `tid` thread state in the final host state is defined to be `upper_timer dschedmax`), while a call to `pselect` with no file descriptors specified and a timeout of 30sec will return at some point in the interval $[30, 30 + \text{dschedmax}]$ 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 [Lynch and Vaandrager 1996, §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 timesteps 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 [Lynch and Vaandrager 1996, §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 below) models time passage, say of duration dur , by evolving each timer in the model state forward by dur . 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 rule. The timers ensure that the specification models the behaviour of real systems with (boundedly) inaccurate clocks: the rate of a host's 'ticker' is constrained only to be within certain bounds of unity.

<p style="margin: 0;"><code>epsilon_1</code> all: misc nonurgent Time passes</p> $h \xrightarrow{dur} h'$ <p style="margin: 0;">let $hs' = \text{Time_Pass_host } dur \ h$ in is_some $hs' \wedge$ $h' \in (\mathbf{the} \ hs')$ \wedge $\neg(\exists rn \ rp \ rc \ lbl \ h'.rn / * \ rp, rc * / h \xrightarrow{lbl} h' \wedge \text{is_urgent } rc)$</p>
--

Description Allow time to pass for dur seconds, updating the internal timers of host h by duration dur to become host state h' . This is only enabled if the host state h is not urgent, i.e. if no τ transition of an urgent rule can fire (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). Notice that, apart from when a timer becomes zero, a host state never becomes urgent due merely to time passage. This means we need only test for urgency at the beginning of the time interval, not throughout it.

Many timed process algebras enforce a *maximal progress* property [Yi 1991], 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.

3.3.4. 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. 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 below for illustration.

<i>accept_1</i>	tcp: succeed	Return new connection; either immediately or from a blocked state.
<i>accept_2</i>	tcp: block	Block waiting for connection.
<i>accept_3</i>	tcp: fail	Fail with EAGAIN: no pending connections and non-blocking semantics set.
<i>accept_4</i>	tcp: fail	Fail with ECONNABORTED: the listening socket has <i>cantsndmore</i> set or has become CLOSED. Returns either immediately or from a blocked state.
<i>accept_5</i>	tcp: fail	Fail with EINVAL: socket not in LISTEN state.
<i>accept_6</i>	tcp: fail	Fail with EMFILE: out of file descriptors.
<i>accept_7</i>	udp: fail	Fail with EOPNOTSUPP or EINVAL: <code>accept()</code> called on a UDP socket.

3.3.5. Relationship Between Code and Specification Structure. In writing the specification we have examined the implementation source code closely, but the two have very different structure. The code is in C, typically with a rough layer structure (but tight coupling between some layers). It has accreted changes over the years, giving a tangled control flow in some parts, and is optimised for fast-path performance. For the specification, however, clarity is the prime concern. Individual rules correspond very roughly to execution *paths* through implementation code.

Each rule is as far as possible declarative, defining a relation between outputs and inputs. In some cases we have been forced to introduce extra structure to mirror oddities in the implementations, e.g. intermediate state variables to record side effects that subsidiary functions have before a segment is dropped, and clauses to model the fact that the BSD fast-path optimisation is not precisely equivalent to the slow path. In developing the specification, we did careful manual unfolding of some of the C control flow to identify such issues; sound refactoring or partial evaluation tools for C would have been very useful at that point.

3.3.6. 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* and *h'*) and of projection and update operations for records and finite maps; the latter are pure functional operations, not imperative updates.

The overall host state structure roughly follows that of the system state: as we saw in §3.2.1, 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.

3.3.7. Concurrency, Blocking Calls, and Atomicity. In the implementations the host state may be modified by multiple threads making concurrent 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 in the implementations we examined 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.

3.3.8. Presentation and Automated Typesetting. We aimed to make the specification as readable as possible, to make it usable as a reference document. The mathematical structure discussed above is important for that, but document presentation also makes a big difference, at many scales:

- local typesetting of expressions;
- structured presentation of transition rules, as seen in Figs. 6 and 7;
- structured presentation of the family of rules for each Sockets API call, such as those for accept above, and each aspect of the wire protocol, with an introductory preamble (not shown here) for each; and
- large-scale organisation of the document.

We paid attention to all of these, building an automated typesetting system, HOLDoc, that takes the HOL source, including annotations for structure and exposition, and outputs LaTeX. This gives a production-quality result without the errors and workflow cost that manual transcription would introduce. The parts of the specification quoted in this document are taken directly or lightly hand-edited from this. The automatically produced polished document was also invaluable during development of the specification, to help keep track of all the details. The HOL source is used to determine the various different kinds of identifiers (types, constructors, auxiliary definitions, and quantified or lambda-bound variables), which are set in appropriate fonts. The tool does not do a full HOL parse, however, so identifiers used at more than one kind are occasionally set wrongly. Importantly, it preserves source-file indentation and (one flavour of) comment, to give easy control of the resulting layout. It has custom support for the top-level structure of the specification, but most of its functionality is general; it has been used for other HOL work and for typesetting unrelated and non-HOL papers. Committing enough effort to do such engineering work to a sufficiently robust standard was essential to make the project as a whole successful.

3.4. Example Trace

Fig. 9 (Page 36) shows an extract from a captured trace with the observed labels for Sockets API calls and returns and TCP segment sends and receives. It is annotated on the left with the sequence of rules *connect_1*, *epsilon_1*, ... used to match this trace when it was checked. Note the time passage transitions (rule *epsilon_1*) and the various internal (τ) steps: *deliver_in_2* dequeuing a SYN,ACK segment and generating an ACK (to be later output by *deliver_out_99*), *connect_2* setting up the

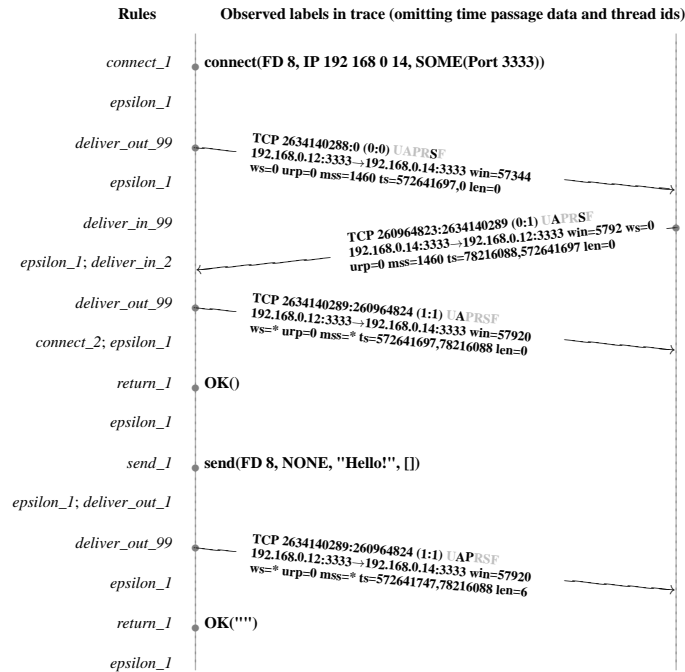


Fig. 9. Extract from sample checked TCP trace, with rule firings.

return from the blocked `connect()`, and `deliver_out_1` enqueueing the “Hello!” segment for output. The diagram shows only the rule names and labels, omitting the symbolic internal state of the host which is calculated at each point. It is automatically generated from the result of checking the trace.

This trace shows a common case, and should be unsurprising for those familiar with TCP. However, the specification covers TCP in *full detail*: fast retransmit and recovery, RTT measurement, PAWS, and so on, and for *all possible inputs*, not just common cases: error behaviour, pathological corners, concurrent socket calls, and so on. Such completeness of specification is an important part of our rigorous approach.

4. SERVICE-LEVEL SPECIFICATION

The previous section described our protocol-level specification, which characterises the on-the-wire behaviour of the TCP protocol in terms of the individual TCP segments exchanged between TCP endpoints, together with the Sockets API and UDP/ICMP behaviour.

We now turn to our *service-level* specification, as introduced in §1.5.2 and shown in Fig. 2 (Page 9). TCP is designed to provide (roughly) a bidirectional reliable byte-stream service between pairs of endpoints, and programmers using TCP above the Sockets API often want to think of it in those terms. Our service-level specification provides such a model, abstracting from the protocol-level details to describe the behaviour of pairs of hosts communicating over TCP, but observed *only* at their Sockets API interfaces. It does not deal with TCP segments on the wire (though it necessarily does include ICMP and UDP messages).

In principle one could derive a service-level specification directly from the protocol model, taking the set of traces it defines and erasing the TCP wire segment transitions. However, that would not give a *usable* specification: one in which key properties of TCP, that users depend on, are clearly visible. Instead, we built the service-level specification by hand, defining a more abstract notion of host state, an abstract notion of stream object, and a new network transition relation, but aiming to give the same Sockets-API-observable behaviour.

The abstract host states are substantially simpler than those of the protocol-level model. For example, the protocol-level TCP control block contains 44 fields, including retransmit and keep-alive timers; window sizes, sequence position and scaling information; timestamping and round trip times. Almost none of these are relevant to the service-level observable behaviour, and so are not needed in the service-level TCP control block. Along with this, the transition rules that define the protocol dynamics, such as *deliver_in_1* and *deliver_in_3*, become much simpler. The rules that deal with the Sockets API must be adapted to the new host state, but they remain largely as before. The overall size of the specification is therefore not much changed.

A naive approach to writing the individual rules would be to existentially quantify those parts of the host state that are missing at the service level (and then to logically simplify as much as possible). However, this would lead to a highly non-deterministic and ultimately less useful specification. Instead, we relied on a number of invariants of the low-level model, arguing informally that, given those, the two behaviours match. We rely on the later validation to detect any errors in these informal arguments.

In the rest of this section we aim to give a flavour of the service-level specification, referring the interested reader to the complete specification online [Ridge et al. 2009; The Netsem Project].

4.1. Streams

The heart of the service-level specification is a model of a bidirectional TCP connection as a pair of unidirectional byte streams between Socket endpoints:

– **unidirectional stream :**

```
tcpStream = (⟦ i : ip;(* source IP *)
              p : port;(* source port *)
              flgs : streamFlags;
              data : byte list;
              destroyed : bool⟧)
```

The *data* in the stream is just a byte list, in contrast to the protocol-level specification combination of byte lists buffered in the sending socket, TCP segments in the sending host's outqueue *oq*, on the wire, and in the receiving host's inqueue *iq*, and bytes buffered in the receiving socket (as in the types we saw in §3.2.1). Further fields record the source IP address and port of the stream, control information in the form of flags, and a boolean indicating whether the stream has been destroyed at the source (say, by deleting the associated socket). Some of these fields are shared with the low-level specification, but others are purely abstract entities. Note that although a stream may be destroyed at the source, previously sent messages may still be on the wire, and might later be accepted by the receiver, so we cannot simply remove the stream when it is destroyed. Similarly, if the source receives a message for a deleted socket, a RST will typically be generated, which must be recorded in the stream flags of the destroyed stream. The following flags record whether the stream is opening (*SYN*, *SYNACK*), closing normally (*FIN*) or abnormally (*RST*).

– **stream control information :**

```
streamFlags = (⟦ SYN : bool;(* SYN, no ACK *)
                SYNACK : bool;(* SYN with ACK *)
                FIN : bool;
                RST : bool⟧)
```

This control information is carefully manually abstracted from the protocol level, to capture just enough structure to express the user-visible behaviour. Note that the *SYN* and *SYNACK* flags may be set simultaneously, indicating the presence of both kinds of message on the wire. The receiver typically lowers the stream *SYN* flag on receipt of a *SYN*: even though messages with a *SYN* may still be on the wire, subsequent *SYN*s will be detected by the receiver as invalid duplicates of the original. A bidirectional stream is then just a pair of unidirectional streams.

The basic operations on a byte stream are to read and write data, though some care is needed for the control information. For example, the following defines a write from Sockets endpoint (i_1, p_1) to endpoint (i_2, p_2) .

– **write flags and data to a stream :**
 $\text{write}(i_1, p_1, i_2, p_2)(flgs, data) s s' = ($
 $\quad \exists in_out in' out'.$
 $\quad \text{sync_streams}(i_1, p_1, i_2, p_2) s(in_out) \wedge$
 $\quad \text{sync_streams}(i_1, p_1, i_2, p_2) s'(in', out') \wedge$
 $\quad in' = in_out \wedge$
 $\quad out'.flgs =$
 $\quad \llbracket SYN := (out.flgs.SYN \vee flgs.SYN);$
 $\quad \quad SYNACK := (out.flgs.SYNACK \vee flgs.SYNACK);$
 $\quad \quad FIN := (out.flgs.FIN \vee flgs.FIN);$
 $\quad \quad RST := (out.flgs.RST \vee flgs.RST) \rrbracket \wedge$
 $\quad out'.data = (out.data ++ data))$

Stream s' is the result of writing $flgs$ and $data$ to stream s . Stream s consists of a unidirectional input stream in_out and output stream out , extracted from the bidirectional stream using the auxiliary sync_streams function. Similarly s' , the state of the stream after the write, consists of in' and out' . Since we are writing to the output stream, the input stream remains unchanged, $in' = in_out$. The flags on the output stream are modified to reflect $flgs$. For example, SYN is in $out'.flgs$ iff $flgs$ contains a SYN or $out.flgs$ already has SYN set. Finally, $out'.data$ is updated by appending $data$ to $out.data$.

4.2. Sample Service-level Transition Rule: $send_3$

Fig. 10 (Page 39) gives the service-level analogue of the $send_3$ protocol-level rule of Fig. 7 (Page 29). The transition occurs between triples $(h \llbracket \dots \rrbracket, S_0 \oplus [\dots], M)$, each consisting of a host, a finite map from stream identifiers to streams, and a set of UDP and ICMP messages. The latter do not play an active part in this rule, and can be safely ignored. Host state is unpacked from the host as before. Note that protocol-level constructs such as $rcvurp$ and $iobc$ are absent from the service-level host state. As well as the host transition, there is a transition of the related stream s to s' . The stream is unpacked from the finite map via its unique identifier $\text{streamid_of_quad}(i_1, p_1, i_2, p_2)$, derived from its quad.

As before, the conditions for this rule require that the state of the socket st must be ESTABLISHED or CLOSE_WAIT. Stream s' is the result of writing string str' and flags $flgs$ to s . Since $flgs$ are all false, the write does not cause any control flags to be set in s' , although they may already be set in s of course.

This rule, and the preceding definitions, demonstrate the conceptual simplicity and stream-like nature of the service level. Other interesting properties of TCP are clearly captured by the service-level specification. For example, individual writes do not insert record boundaries in the byte stream, and in general a read returns only part of the data, uncorrelated with any particular write. The model also makes clear that the unidirectional streams are to a large extent independent. For example, closing one direction does not automatically cause the other to close.

5. ABSTRACTION FUNCTION FROM PROTOCOL-LEVEL TO SERVICE-LEVEL STATES

While the service specification details *what* service an implementation of TCP provides to the Sockets interface, our abstraction function details *how* the protocol-level description of the protocol provides that service. The abstraction function maps protocol-level states and transitions to service-level states and transitions. A protocol-level network consists of a set of hosts, each with their own

$send_3$ **tcp: slow nonurgent succeed** Successfully return from blocked state having sent data

$$\begin{aligned}
& (h \{ts := ts \oplus (tid \mapsto (\mathbf{SEND2}(sid, *, str, opts))_d)\}; \\
& \quad socks := socks \oplus [(sid, \mathbf{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore, \\
& \quad \quad \quad \mathbf{TCP_Sock}(st, cb, *)))], \\
& \quad S_0 \oplus [(streamid_of_quad(i_1, p_1, i_2, p_2), s)], M) \\
\overset{\tau}{\rightarrow} & (h \{ts := ts \oplus (tid \mapsto (\mathbf{RET}(\mathbf{OK}(\mathbf{implode} \mathit{str}''))_{sched_timer}))\}; \\
& \quad socks := socks \oplus [(sid, \mathbf{SOCK}(\uparrow fid, sf, \uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2, *, \mathbf{F}, cantrcvmore, \\
& \quad \quad \quad \mathbf{TCP_Sock}(st, cb, *)))], \\
& \quad S_0 \oplus [(streamid_of_quad(i_1, p_1, i_2, p_2), s')], M)
\end{aligned}$$

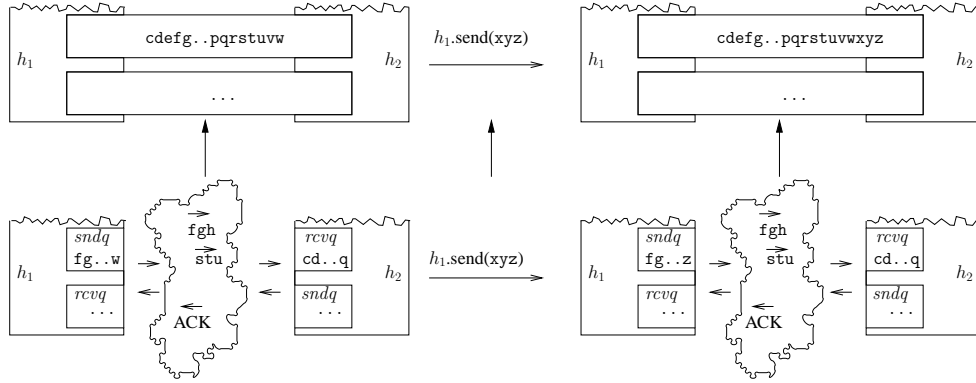
$$\begin{aligned}
& st \in \{\mathbf{ESTABLISHED}; \mathbf{CLOSE_WAIT}\} \wedge \\
& space \in \mathbf{UNIV} \wedge \\
& space \geq \mathbf{length} \mathit{str} \wedge \\
& str' = str \wedge str'' = [] \wedge \\
& flgs = flgs \langle \mathbf{SYN} := \mathbf{F}; \mathbf{SYNACK} := \mathbf{F}; \mathbf{FIN} := \mathbf{F}; \mathbf{RST} := \mathbf{F} \rangle \wedge \\
& \mathbf{write}(i_1, p_1, i_2, p_2)(flgs, str')s \ s'
\end{aligned}$$

Description Thread tid is blocked in state $\mathbf{SEND2}(sid, *, str, opts)$ where the TCP socket sid has binding quad $(\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2)$, has no pending error, is not shutdown for writing, and is in state $\mathbf{ESTABLISHED}$ or $\mathbf{CLOSE_WAIT}$.

The space in the socket's send queue, $space$ (calculated using $\mathbf{send_queue_space}$), is greater than or equal to the length of the data to be sent, str . The data is appended to the socket's send queue and the call successfully returns the empty string. A τ transition is made, leaving the thread state $\mathbf{RET}(\mathbf{OK}(\mathbf{implode} \mathit{str}''))$. If the data was marked as out-of-band, $\mathbf{MSG_OOB} \in opts$, then the socket's urgent pointer will be updated to point to the end of the socket's send queue.

Model details The data to be sent is of type string in the $\mathbf{send}()$ call but is a byte list when the datagram is constructed. Here the data, str is of type byte list and in the transition $\mathbf{implode} \mathit{str}$ is used to convert it into a string.

Fig. 10. Sample service-level specification transition rule: $send_3$



The bottom half of the diagram shows a protocol-level transition of host h_1 performing a $send("xyz")$ call in the context of a network with three datagrams in transit and a host h_2 on the other end of the connection. The top half of the diagram shows how those states, and the transition, are mapped into the service-level model by the abstraction function.

Fig. 11. Abstraction function, illustrated (data part only)

TCP stacks, and TCP segments (and UDP and ICFP datagrams) on the wire. The abstraction function takes this data and calculates abstract byte streams between Sockets API endpoints, together with the abstract connection status information.

The latter is the more intricate part, but we give only a simple example here: the service-level *destroyed* flag is set iff either there is no socket on the protocol-level host matching the quad for the TCP connection or the state of the TCP socket is CLOSED.

The former is illustrated in Fig. 11 (Page 40). For example, consider the simple case where communication has already been established, and the source is sending a message to the destination that includes the string "abc...xyz", of which bytes up to "w" have been moved to the source *sndq*. Moreover, the destination has acknowledged all bytes up to "f", so that the *sndq* contains "fgh...uvw", and *snd_una* points to "f". The destination *rcvq* contains "cde...opq", waiting for the user to read from the socket, and *rcv_nxt* points just after "q".

	\downarrow <i>snd_una</i> \downarrow <i>rcv_nxt</i>
message	...abcdefghijklmnopqrstuvwxyz...
source <i>sndq</i>	fghijklmnopqrstuvw
destination <i>rcvq</i>	cdefghijklmnopq
DROP ($rcv_nxt - snd_una$) <i>sndq</i>	rstuvw
stream	cdefghijklmnopqrstuvw

The data that remains in the stream waiting for the destination endpoint to read, is the byte stream "cdefghijklmnopqrstuvw". This is simply the destination *rcvq* with part of the source *sndq* appended: to avoid duplicating the shared part of the byte sequence, $(rcv_nxt - snd_una)$ bytes are dropped from *sndq* before appending it to *rcvq*.

An excerpt from the HOL definition appears in Fig. 12 (Page 41). It takes a quad (i_1, p_1, i_2, p_2) identifying the TCP connection, a source host h , a set of messages *msgs* on the wire, and a destination host i , and produces a unidirectional stream. It follows exactly the previous analysis: $(rcv_nxt - snd_una)$ bytes are dropped from *sndq* to give *sndq'*, which is then appended to *rcvq* to give the data in the stream.

Note that, in keeping with the fact that TCP is designed so that hosts can retransmit any data that is lost on the wire, this abstraction does not depend on the data in transit — at least for normal connections in which neither endpoint has crashed.


```

– unidirectional abstraction function :
abs_hosts_one_sided( $i_1, p_1, i_2, p_2$ )( $h, msgs, i$ ) = (
  (* messages that we are interested in, including  $oq$  and  $iq$  *)
  let ( $hoq, iiq$ ) =
    case ( $h.oq, i.iq$ ) of (( $msgs$ )1, ( $msgs'$ )2) → ( $msgs, msgs'$ ) in
  let  $msgs = \text{list\_to\_set } hoq \cup msgs \cup (\text{list\_to\_set } iiq)$  in
  (* only consider TCP messages ... *)
  let  $msgs = \{msg \mid TCP \ msg \in msgs\}$  in
  (* ... that match the quad *)
  let  $msgs = msgs \cap$ 
    { $msg \mid msg = msg \llbracket is_1 := \uparrow i_1; ps_1 := \uparrow p_1; is_2 := \uparrow i_2; ps_2 := \uparrow p_2 \rrbracket$ } in

  (* pick out the send and receive sockets *)
  let  $smatch \ i_1 \ p_1 \ i_2 \ p_2 \ s =$ 
    (( $s.is_1, s.ps_1, s.is_2, s.ps_2$ ) = ( $\uparrow i_1, \uparrow p_1, \uparrow i_2, \uparrow p_2$ )) in
  let  $snd\_sock = \text{Punique\_range}(smatch \ i_1 \ p_1 \ i_2 \ p_2)h.socks$  in
  let  $rcv\_sock = \text{Punique\_range}(smatch \ i_2 \ p_2 \ i_1 \ p_1)i.socks$  in
  let  $tcpsock\_of \ sock = \text{case } sock.pr \ \text{of}$ 
     $TCP1\_hostTypes \ \$TCP\_PROTO \ tcpsock \rightarrow tcpsock$ 
     $\parallel \_3 \rightarrow \text{ERROR"abs\_hosts\_one\_sided:tcpsock\_of"}$ 
  in
  (* the core of the abstraction function is to compute  $data$  *)
  let ( $data : \text{byte list}$ ) = case ( $snd\_sock, rcv\_sock$ ) of
    ( $\uparrow(\_8, hsock), \uparrow(\_9, isock)$ ) → (
      let  $htcpsock = tcpsock\_of \ hsock$  in
      let  $itcpsock = tcpsock\_of \ isock$  in
      let ( $snd\_una, sndq$ ) = ( $htcpsock.cb.snd\_una, htcpsock.sndq$ ) in
      let ( $rcv\_nxt, rcvq$ ) = ( $itcpsock.cb.rcv\_nxt, itcpsock.rcvq$ ) in
      let  $rcv\_nxt = tcp\_seq\_flip\_sense \ rcv\_nxt$  in
      let  $sndq' = \text{DROP}((\text{num}(rcv\_nxt - snd\_una)))sndq$  in
       $rcvq ++ sndq'$ 

       $\parallel (\uparrow(\_8, hsock), *) \rightarrow$ 
        let  $htcpsock = tcpsock\_of \ hsock$  in
         $htcpsock.sndq$ 

       $\parallel (*, \uparrow(\_9, isock)) \rightarrow$ 
        let  $itcpsock = tcpsock\_of \ isock$  in
        let ( $rcv\_nxt : tcpLocal \ seq32, rcvq : \text{byte list}$ ) =
          ( $tcp\_seq\_flip\_sense(itcpsock.cb.rcv\_nxt), itcpsock.rcvq$ ) in
           $rcvq ++ (\text{stream\_reass } rcv\_nxt \ msgs)$ 

       $\parallel (*, *) \rightarrow \text{ERROR"abs\_hosts\_one\_sided:data"}$ 
    in
     $\llbracket i := i_1;$ 
     $p := p_1;$ 
     $flgs :=$ 
     $\llbracket SYN := (\exists msg.msg \in msgs \wedge msg = msg \llbracket SYN := \mathbf{T}; ACK := \mathbf{F}$ );
     $SYNACK := (\exists msg.msg \in msgs \wedge msg = msg \llbracket SYN := \mathbf{T}; ACK := \mathbf{T}$ );
     $FIN := (\exists msg.msg \in msgs \wedge msg = msg \llbracket FIN := \mathbf{T}$ );
     $RST := (\exists msg.msg \in msgs \wedge msg = msg \llbracket RST := \mathbf{T}$ );
     $\rrbracket;$ 
     $data := data;$ 
     $destroyed := (\text{case } snd\_sock \ \text{of}$ 
     $\uparrow(sid, hsock) \rightarrow ((tcpsock\_of \ hsock).st = \text{CLOSED})$ 
     $\parallel * \rightarrow \mathbf{T}$ 
     $\rrbracket)$ 
  )

```

Fig. 12. Abstraction function, excerpt

For a given TCP connection, the full abstraction function uses the unidirectional function twice to form a bidirectional stream constituting the service-level state. As well as mapping the states, the abstraction function maps the transition labels. Labels corresponding to visible actions at the Sockets interface, such as a `connect` call, map to themselves. Labels corresponding to internal protocol actions, such as the host network interface sending and receiving datagrams from the wire, are invisible at the service level, and so are mapped to τ , indicating no observable transition. Thus, for each protocol-level transition, the abstraction function gives a service-level transition with the same behaviour at the Sockets interface. Mapping the abstraction function over a protocol-level trace gives a service-level trace with identical Sockets behaviour. Every valid protocol-level trace should map to a valid service-level trace.

6. EXPERIMENTAL VALIDATION: TESTING INFRASTRUCTURE

We now turn to the problem of testing and assessing the consistency between TCP implementations (written in C), our protocol-level model (in HOL), and our service-level specification (also in HOL).

As introduced in §1.5, we did this with an experimental semantics process. For the relationship between implementations and our protocol-level specification:

- we wrote a large number of tests (using automation to combinatorially test a wide range of cases);
- we instrumented production implementations to record traces of the events corresponding to those of our protocol-level specification interfaces (as described in §3.1);
- we built a special-purpose checker within HOL4 that checks whether an experimental trace is admitted by the specification; and
- we built infrastructure to support this testing and checking at scale.

For the relationship between the protocol-level and service-level specification, we recorded (a smaller number of) double-ended traces, and built a system that let us check that the abstraction function of §5 relates protocol-level and service-level states at each step along the trace.

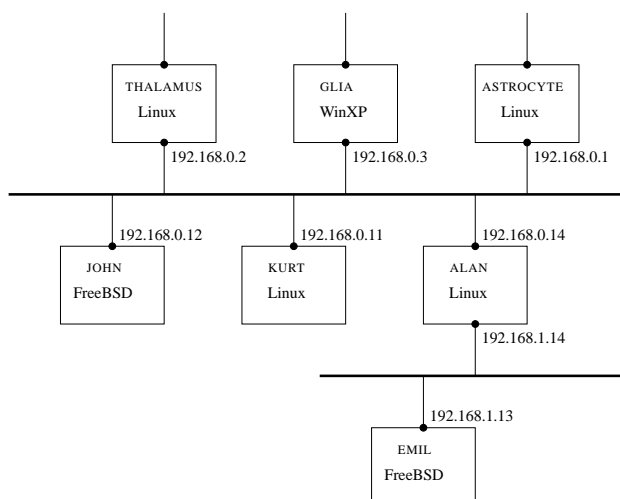
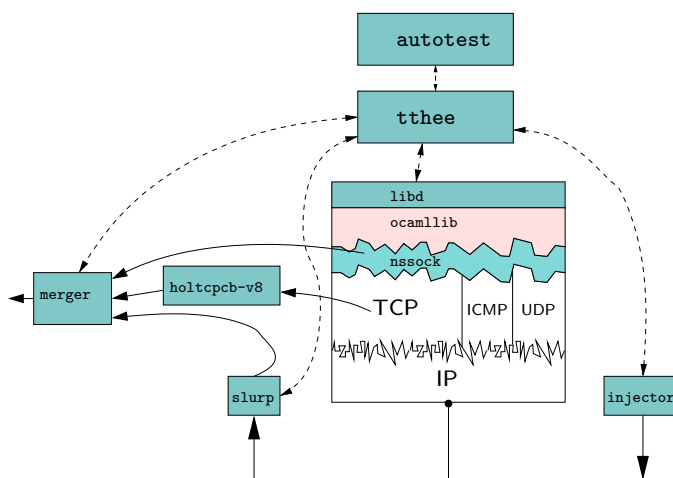
Recall that for TCP the implementations are the de facto standard. In producing specifications after the fact, we aim to validate the specifications against the implementation behaviour, but this machinery could equally well be used in the other direction for new implementations.

We begin in this section with the systems work needed for this: instrumentation, test generation, distributed trace checking, and data visualisation. The theorem-prover work of our trace checkers is described in §7, and the experimental results are in §8.

6.1. Experimental Setup and Instrumentation for Trace Generation

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. 13 (Page 43). We instrument the wire interface with a `slurp` tool above the standard `libpcap`, instrument the Sockets API with an `nssock` wrapper, and on BSD additionally capture TCP control block records generated by the `TCP_DEBUG` kernel option. All three produce HOL format records which are merged into a single trace; this requires accurate timestamping, with careful management of NTP offsets between machines and propagation delays between them. Our abstractions from actual system behaviour to specification-interface observables, as discussed in §1.7 and §3.1, are implemented in these tools. For example, `slurp` performs reassembly of IP fragments into the TCP datagrams we saw in Fig. 4 (Page 22), pulls out the TCP datagram options represented there, and so on, to produce values of that `tcpSegment HOL4` type.

A test executive `tthree` drives the system by making Sockets API calls (via a `libd` daemon) and directly injecting messages with an `injector` tool. These tools are written in OCaml [Leroy et al. 2004] with additional C libraries. 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 (as in Fig. 9 (Page 36)).



A sample configuration of our instrumentation and test generation tools, for a single host under test, and one of our test networks.

Fig. 13. Testing infrastructure.

For the service-level validation, we began with a similar instrumented test network, but collected double-ended traces, capturing the behaviour of two interacting hosts, rather than just one endpoint.

6.2. Tests

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 wrote tests to, as far as we could, exercise all the interesting behaviour of the protocols and API, with manually written combinatorial generation. Almost all tests were run on all three OSs; many are automatically iterated over a selection of TCP socket states, port values, etc. In total around 6000 traces are generated.

For example, trace 1484, of intermediate complexity, is informally described as follows: “send() – for a non-blocking socket in state ESTABLISHED (*NO_DATA*), with a reduced send buffer that is almost full, attempt to send more data than there is space available.”

Assessing coverage of the traces is non-trivial, as the system is essentially infinite-state, but we can check that almost all the host LTS rules are covered. The traces are relatively short, so they may not exercise some of TCP’s congestion-control regimes.

6.3. Distributed Checking Infrastructure

For good coverage we want to check many traces, and this had to be repeated often during development of the specification. Such checking, using the theorem-prover infrastructure described in §7, is computationally intensive but naturally parallel: each trace (apart from initialisation of the evaluator) is independent. We therefore distributed checking over as many processors as possible.

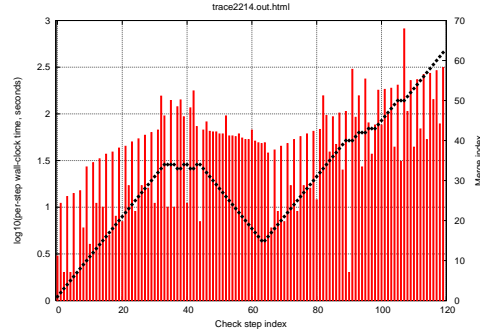
Checking is compute-bound, not space- or IO-limited. A typical trace check run might require 100MB of memory (a few need more); most trace input files are only of the order of 10KB, and the raw checker output for a trace is 100KB – 3MB. We used approximately 100 processors, running background jobs on personal workstations and lab machines (the fastest being dual 3.06GHz Xeons) and using a processor bank of 25 dual Opteron 250s. We relied on a common NFS-mounted file system. Note that these machines were typical workstation and server machines of around 2004; current hardware would provide significantly greater performance, and there have also been substantial improvements in HOL4 performance.

Achieving satisfactory performance of the symbolic evaluator was critical for this work to be feasible, and needed considerable work: algorithmic improvements to HOL itself (e.g. in the treatment of multi-field records), to the evaluator (e.g. in better heuristics for search, and the lazy computation and urgency approximations mentioned in §7), and to the checking infrastructure, distributing over more machines and using them more efficiently. These reduced the total TCP check time from 500 hours, which was at the upper limit of what was practical. By the end of the project, checking around 2600 UDP traces took approximately 5 hours, which is perfectly usable. For TCP the checker has a much more complex task. TCP host states are typically more symbolic, with more fields that are only loosely constrained and with larger sets of constraints. Also, longer traces are required to reach the various possible states of the system. Checking a complete run (around 1100 traces) of the BSD traces took around 50 hours, which is manageable if not ideal.

Fig. 14 (Page 45) suggests the checker run-time per step rises piecewise exponentially with the trace length, though with a small exponent. This is due to the gradual accumulation of constraints, especially time passage rate constraints. In principle there is no reason why in long traces they could not be agglomerated.

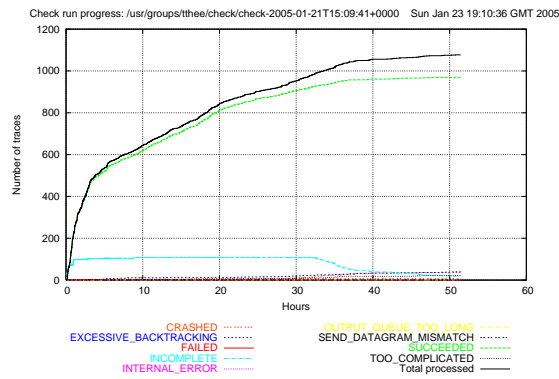
6.4. Visualisation Tools and Data Management

The resulting datasets are large and good visualisation tools are necessary for working with them. Our main tool was an HTML display of the results of each check run, with, for each trace, a link to the checker output, the trace in HTML and graphical form (as in Fig. 9 (Page 36)), the short description, and a graph showing the backtracking and progress of the checker (as in Fig. 14 (Page 45)). To help us manage the outstanding issues during development of the specification, we maintained a file of trace annotations, identifying subsets of the traces that have not succeeded for some particular reason and indicating whether that problem should have been resolved. The display shows the expected and actual number of successes for these. The progress of a whole run can be visualised as in Fig. 15 (Page 45), useful to determine when best to abort an existing run in order to restart with an improved checker or specification. Figure 16 shows the progress of a check run indexed by the trace number and trace length, useful for seeing patterns of non-successes.



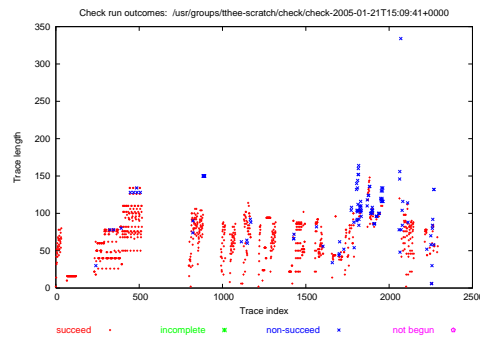
Bars indicate the checker execution time for each step, on the left scale. Diamonds indicate how far through the trace each step is, on the right scale. This trace, atypically, required significant backtracking; most need no backtracking of depth greater than one.

Fig. 14. Checker monitoring: timed step graph.



This indicates how an entire check run progressed, showing the number of traces processed, succeeded, and non-succeeded for various reasons. The INCOMPLETE line (dropping around 33 hours) indicates roughly how many worker machines were active. The shape of the graph is largely determined by the order of traces in the run, shortest first.

Fig. 15. Checker monitoring: progress of a TCP run.



This indicates how a TCP check runs progressed, showing the traces processed, succeeded, and non-succeeded for various reasons, indexed by the trace number and trace length.

Fig. 16. Checker monitoring: progress of a UDP and a TCP run.

Our experience was that devoting sufficient engineering effort to building and refining all these tools, as well as those for automating testing and checking, was essential.

We also built an explicit regression tester, comparing the results of multiple check runs (which might be on overlapping but non-identical trace sets), but did not use it heavily — the annotation display was more useful, especially as we reached closer to 100% success.

7. EXPERIMENTAL VALIDATION: CHECKING TECHNOLOGY

Our computational task for checking an observed endpoint trace against our protocol-level specification 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 $l_1 \dots l_n$, determine whether h_0 can exhibit this behaviour in the model. The transition system includes unobservable τ labels, so we actually have to determine whether there is a sequence

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

for some 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. 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); and
- 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 with branching (indeed, for many types there are an infinite number of possibilities). Instead, the system maintains sets of constraints (which are just arbitrary HOL predicates), attached to each transition. These constraints are simplified (including the action of arithmetic decision procedures) and checked for satisfiability as checking proceeds. Later labels often fully determine variables that were introduced earlier, e.g. for file descriptors, TCP options, etc.

For example, a *connect_1* transition in one particular TCP trace, modelling the `connect()` invocation, introduced new variables:

```
(advms :num), (advms' :num option),
(cb'_2_rcv_wnd :num), (n :num), (rcv_wnd0 :num),
(request_r_scale :num option), (ws :char option)
```

and new constraints:

```
∀n2. advms' = SOME n2 ==> n2 <= 65535
∀n2. request_r_scale = SOME n2 ==> ORD (THE ws) = n2
pending (cb'_2_rcv_wnd=rcv_wnd0* 2**case 0 I request_r_scale)
```

```

pending (ws = OPTION_MAP CHR request_r_scale)
advms <= 65495
cb'_2_rcv_wnd <= 57344
n <= 5000
rcv_wnd0 <= 65535
1 <= advms
1 <= rcv_wnd0
1024 <= n
advms' = NONE ∨ advms' = SOME advms
request_r_scale=NONE ∨ ∃n1.request_r_scale=SOME n1 ∧ n1<=14
nrange n 1024 3976
nrange rcv_wnd0 1 65534
case ws of NONE -> T || SOME v1 -> ORD v1 <= TCP_MAXWINSCALE

```

Many of these constraints are numeric (over various different numeric types), but some are more complex. For example, the above includes option-type and record operations, with some nested quantifiers. In other cases there is potential nondeterminism arising from the multiple ways in which the data from multiple TCP segments, with overlapping sequence numbers, can be assembled into a single stream.

Hence, instead of finding a sequence of theorems of the form

$$\begin{aligned}
&\vdash h_0 \xrightarrow{l_1} h_1 \\
&\vdash h_1 \xrightarrow{l_2} h_2 \\
&\dots \\
&\vdash h_{n-1} \xrightarrow{l_n} h_n
\end{aligned}$$

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

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

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 nondeterministically, 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

$$\begin{array}{c} \Gamma, p \vee q \vdash h_{i-1} \xrightarrow{l_i} h_i \\ \swarrow \quad \searrow \\ \Gamma, p \vdash h_{i-1} \xrightarrow{l_i} h_i \quad \Gamma, q \vdash h_{i-1} \xrightarrow{l_i} h_i \end{array}$$

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

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

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

7.1. 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 $\langle\langle fld_1 := v_1; fld_2 := v_2; \dots \rangle\rangle$), 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

$$\langle\langle fld_1 := v_1; fld_2 := v_2; \dots \rangle\rangle \xrightarrow{l} h \equiv (D_1 \vee \dots \vee D_{n-1} \vee D_n) \quad (1)$$

where l is a label form (such as $(tid\text{-}socket\ (arg))$ or τ or \overline{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 $h = \langle\langle fld_1 := v'_1; fld_2 := v'_2; \dots \rangle\rangle$, 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

$$\vdash D_i \Rightarrow \langle\langle fld_1 := v_1; \dots \rangle\rangle \xrightarrow{l_0} \langle\langle fld_1 := v'_1; \dots \rangle\rangle$$

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

$$\Gamma_0, D'_i \vdash \langle \langle f d_1 := v_1; \dots \rangle \rangle \xrightarrow{I_0} \langle \langle f d_1 := v'_1; \dots \rangle \rangle$$

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.

7.2. 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 non-ground components. It is important in this setting to retain variable bindings rather than simply substituting them out. Substituting non-ground 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

$$x = E \vdash \dots (\text{if } x = [] \text{ then } f(x) \text{ else } g(x)) \dots$$

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

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

In this situation, the application of g to a list known to 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.

7.3. 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*, shown in §3.3.3, which allows time to pass if the host state is not urgent. 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.

7.4. 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

$$h \langle \langle fld_1 := v_0 \rangle \rangle \xrightarrow{l} h \langle \langle fld_1 := v \rangle \rangle$$

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

$$\begin{aligned} & \langle \langle fld_1 := v_0; fld_2 := v_2; fld_3 := v_3; \dots \rangle \rangle \\ \xrightarrow{l} & \langle \langle fld_1 := v; fld_2 := v_2; fld_3 := v_3; \dots \rangle \rangle \end{aligned}$$

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

$$\begin{array}{c} \langle ts := tidmap \oplus (t, \text{RUN}); \dots \rangle \\ \xrightarrow{l} \\ \langle ts := tidmap \oplus (t, \text{newstate}); \dots \rangle \end{array}$$

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

$$\begin{array}{c} \langle ts := tidmap; \dots \rangle \\ \xrightarrow{l} \\ \langle ts := tidmap \oplus (t, \text{newstate}) \rangle \end{array}$$

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

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

$$\begin{array}{l} \text{fmScan } \emptyset \ k_2 \ v_2 = \perp \\ \text{fmScan } (fm \oplus (k_1, v_1)) \ k_2 \ v_2 = (k_1 = k_2 \wedge v_1 = v_2) \vee \\ \text{fmScan } (fm \setminus k_1) \ k_2 \ v_2 \end{array}$$

where $fm \setminus k$ 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 $\text{close_}\mathcal{I}$, 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

$$\begin{array}{c} \langle \text{socks} := \text{sockmap} \oplus (\text{sid}, \text{sock}); \dots \rangle \\ \xrightarrow{\text{tid.close } (fd)} \\ \langle \text{socks} := \text{sockmap}; \dots \rangle \end{array}$$

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

$$\frac{\langle\langle socks := sockmap; \dots \rangle\rangle}{\begin{array}{c} \text{tid}\cdot\text{close}(fd) \\ \longrightarrow \\ \langle\langle socks := sockmap \setminus sid; \dots \rangle\rangle \end{array}}$$

$$\text{fmscan } sockmap \ sid \ sock \ \wedge$$

$$\text{sideconditions}[sockmap := sockmap \setminus sid]$$

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 \setminus 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 `local_ips` is defined:

$$\text{local_ips}(ifmap) = \bigcup_{(k,i) \in ifmap} i.ipset$$

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

$$\begin{aligned} \text{local_ips}(\emptyset) &= \emptyset \\ \text{local_ips}(ifmap \oplus (k, i)) &= i.ipset \cup \text{local_ips}(ifmap \setminus k) \end{aligned}$$

is used instead.

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

$$\begin{aligned} R \ x \ y = \exists \vec{v}. \text{let } u_1 = e_1 \text{ in} \\ \text{let } u_2 = e_2 \text{ in} \\ \dots \\ c_1 \wedge c_2 \wedge c_3 \wedge \dots \wedge c_n \end{aligned}$$

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 `tcp_reass`. Involving two gruesome set-comprehensions, `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.)

7.5. Adding Constraints

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

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

We do sometimes add constraints that are consequences of existing assumptions, which 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.

7.6. 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 \perp .) 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 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

$$s_1 \triangleleft s_2 \equiv s_1 - s_2 \triangleleft 0$$

where \triangleleft 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.

7.7. Service-level Validation

For the service-level validation, we started with double-ended traces, describing the behaviour of two interacting hosts rather than just one endpoint. We then used our previous symbolic evaluation tool to discover symbolic traces of the protocol-level model that corresponded to the real-world traces. That is a complex and computationally intensive process, involving the backtracking depth-

first search and constraint simplification described above, essentially to discover internal host state and internal transitions that are not explicit in the trace.

We then *ground* these symbolic traces, finding instantiations of their variables that satisfy any remaining constraints, to produce a ground protocol-level trace in which all information is explicit. Given such a ground trace, we can map the abstraction function over it to produce a candidate ground service-level trace.

It is then necessary to check validity of this trace, which is done with a service-level test oracle. As at the protocol level, we wrote a new special-purpose service-level checker in HOL which performs symbolic evaluation of the specification with respect to ground service-level traces. Crucially, this checking process is much simpler than that at the protocol level because all host values, and all transitions, are already known. All that remains is to check each ground service-level transition against the specification.

The most significant difference between the old and new checkers is that the former had to perform a depth-first search to even determine which rule of the protocol model was appropriate. Because that work has already been done, and because the two specifications have been constructed so that their individual rules correspond, the service-level checker does not need to do this search. Instead, it can simply check the service-level version of the rule that was checked at the protocol level, dealing with each transition in isolation. In particular, this means that the service-level checker need not attempt to infer the existence of unobservable τ -transitions.

Another significant difference between the two checkers is that the service-level checker can aggressively search for instantiations of existentially quantified variables that arise when a rule's hypothesis has to be discharged. At the protocol level, such variables may appear quite unconstrained at first appearance, but then become progressively more constrained as further steps of the trace are processed.

For example, a simplified rule for the `socket` call might appear as

$$\frac{fd \notin \text{usedfds}(h_0)}{h_0 \langle \text{socks} := \text{socks} \rangle \xrightarrow{\text{tid.socket}()} h_0 \langle \text{socks} := \text{socks} \oplus (\text{sid}, fd) \rangle}$$

stating that when a `socket` call is made, the host h_0 's `socks` map is updated to associate the new socket (identified by `sid`) with file-descriptor `fd`, subject only to the constraint that the new descriptor not already be in use. (This under-specification is correct on Windows; on Unix, the file-descriptor is typically the next available natural number.)

In the protocol-level checker, the `fd` variable must be left uninstantiated until its value can be deduced from subsequent steps in the trace. In the service-level checker, both the initial host and the final host are available because they are the product of the abstraction function applied to the previously generated, and ground, protocol trace. In a situation such as this, the variable from the hypothesis is present in the conclusion, and can be immediately instantiated.

In other rules of the service-level specification, there can be a great many variables that occur only in the hypothesis. These are existentially quantified, and the checker must determine if there is an instantiation for them that makes the hypothesis true. The most effective way of performing this check is to simplify, apply decision procedures for arithmetic, and to then repeatedly case-split on boolean variables, and the guards of if-then-else expressions to search for possible instantiations.

The above process is clearly somewhat involved, and itself would ordinarily be prone to error. To protect against this, as for the protocol-level work, we built all the checking infrastructure within HOL. So, when checking a trace, we are actually building machine-checked proofs that its transitions are admitted by the inductive definition of the transition relation in the specification.

8. EXPERIMENTAL VALIDATION: RESULTS

8.1. Protocol-Level Checking 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 focussed 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 UDP non-successes: 36 are due to a problem in test generation (difficulties with accurate timestamping on WinXP); 27 are tests which involve long data strings for which we hit a space limitation of the HOL string library (which uses a particularly non-space-efficient representation at present); 11 are because of known problems with test generation; and 3 are due to an ICMP delivery problem on FreeBSD.

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 due to a collection of 20 issues in the specification which we roughly diagnosed but did not fix, simply for lack of staff resource at the time.

Much of the TCP development was also carried out for all three implementations, and the specification does identify various differences between them. In the later stages we focussed on BSD, for two reasons: the BSD debug trace records make automated validation easier in principle, and as a small research team we had only rather limited staff resources available. We believe that extending the TCP work to fully cover the other implementations would require little in the way of new techniques.

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 should 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 not yet succeeded in validation. Moreover, test generation was largely independent of the validation process (some additional tests were constructed during validation, and some particularly long traces were excluded). For TCP, however, it would be good to check more medium-length traces, to be sure that the various congestion-control regimes are fully explored; our trace set is weighted more towards connection setup/teardown and Sockets API issues. It would also be desirable to check implementation code coverage of our tests, for the FreeBSD and Linux implementations; we did not attempt that.

8.2. Implementation Anomalies

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 the *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, some of which are certainly bugs in the conventional sense. There are explicit OS version dependencies on around 260 lines of the specification, and the report [Bishop et al. 2005b] details around 30 anomalies. 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. We list a few briefly below, mostly for BSD TCP.

- The receive window is updated on receipt of a bad segment.
- Simultaneous open can respond with an ACK rather than a SYN,ACK.

- The code has an erroneous definition of the `TCP_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`.

Many of these oddities, and many of the 260 OS differences, were discovered by our testing process; by describing them we hope primarily to give some sense of what kind of fine-grain detail can be captured by our automated testing process, in which window values, time values, etc. are checked against their allowable ranges as soon as possible. The remainder were found directly in the source code while writing the specification. The main point we observe in the implementations is that their behaviour is extremely complex and irregular, but that is not subject to any easy fix.

8.3. Service-level Results

Our protocol-level validation involved several thousand traces designed to exercise the behaviour of single endpoints, covering both the Sockets API and the wire behaviour. To produce a reasonably accurate specification, we iterated the checking and specification-fixing process many times.

For the service-level specification, we have not attempted the same level of validation, simply due to resource constraints. Instead, we have focused on developing the method, doing enough validation to demonstrate its feasibility. Producing a specification in which one should have high confidence might require another man-year or so of testing and specification improvement — perfectly feasible, and a tiny amount of effort in terms of industrial protocol stack development, but unlikely to lead to new research insights. That said, most of the Sockets API behaviour does not relate to the protocol dynamics and is common between the two specifications, so is already moderately well tested. In all, 30 end-to-end tests were generated, covering a variety of connection setup and tear-down cases and end-to-end communication, but not including packet loss, reordering, duplication, or severe delay. After correcting some specification errors, all these traces were found to validate successfully.

To illustrate again how discriminating our testing process is, we mention two errors we discovered during service-level validation. At the protocol-level, a TCP message moving from a host output queue to the wire corresponds to an unobservable τ event at the service level. Naively we assumed the host state would be unchanged, since the output queue at the service-level carries only ICMP and UDP messages. However, this is not correct, since the transmission of a TCP message alters the timer associated with the output queue, increasing its value. The update to the timer permits the host to delay sending the ICMP and UDP messages. Without this side-effect, the service-level specification effectively required ICMP and UDP messages to be sent earlier than they would otherwise have been. To correct this error, the service specification had to allow the timer to be updated if at the protocol-level there was potentially a TCP message on the queue that might be transferred to the wire. Another error arose in the definition of the abstraction function. The analysis of the merging of the send and receive queues on source and destination hosts, described in Section 5, was initially incorrect, leading to streams with duplicated, or missing, runs of data. Fortunately this error was easy to detect by examining the ground service-level trace, where the duplicated data was immediately apparent.

8.4. Validating the Other Direction

Our validation processes check that certain traces are included in the protocol-level or service-level specification. As we have seen, this can be a very discriminating test, but it does not address the question of whether the specifications admit too many traces. That cannot be determined by refer-

ence to the de facto standard implementations, as a reasonable specification here must be looser than any one implementation. Instead, one must consider whether the specifications are strong enough to be useful, for proving properties of applications that use the Sockets API, or as a basis for new implementations. We mention work of both kinds in §1.9, but more could be done here.

In particular, one could modularly refine the specification, resolving the points at which it is non-deterministic, so that it does describe an implementation. This would entail specifying aspects of the host scheduling (resolving nondeterminism between multiple rules that can fire simultaneously), giving algorithms for choosing initial sequence numbers, options, etc., and constraining TCP output so that it does have the ACK-clock behaviour. It should then be possible to integrate the specification, our symbolic evaluation engine, and the packet injector and slurp tools, to form a working TCP implementation — for example, on receiving a segment from the slurp tool, it would run the symbolic evaluator to calculate a new host state, which might produce new segments to output via the injector. One could gain additional confidence in the validity of the specification by checking this interoperates with existing TCP/IP stacks, though they would have to be artificially slowed down to match the speed of the evaluator. This would demonstrate that executable prototype implementations of future protocols could be directly based on similar specifications.

9. REVISITING THE TCP STATE DIAGRAM

TCP is often presented using a state diagram to describe how the `tcpstate` component of a TCP socket state (CLOSED, LISTEN, ESTABLISHED, etc.) changes with API calls and segments sent and received. The original RFC793 contained one such diagram, and another version (redrawn from Stevens [Stevens 1994; Wright and Stevens 1995; Stevens 1998]) is shown at the top of Fig. 17 (Page 58). This can be useful, explaining in broad terms how the SYN, ACK, and FIN flags in TCP segments are used in connection establishment and teardown, though one has to understand that the `tcpstate` component is only a tiny part of the complete socket state that the protocol endpoint behaviour depends on (see the model types `socket`, `tcp_socket`, and `tcpcb` in §3.2.1, which define the entire socket state, and how much of the `deliver_in_1` rule of Fig. 8 (Page 31) relates to non-`tcpstate` behaviour).

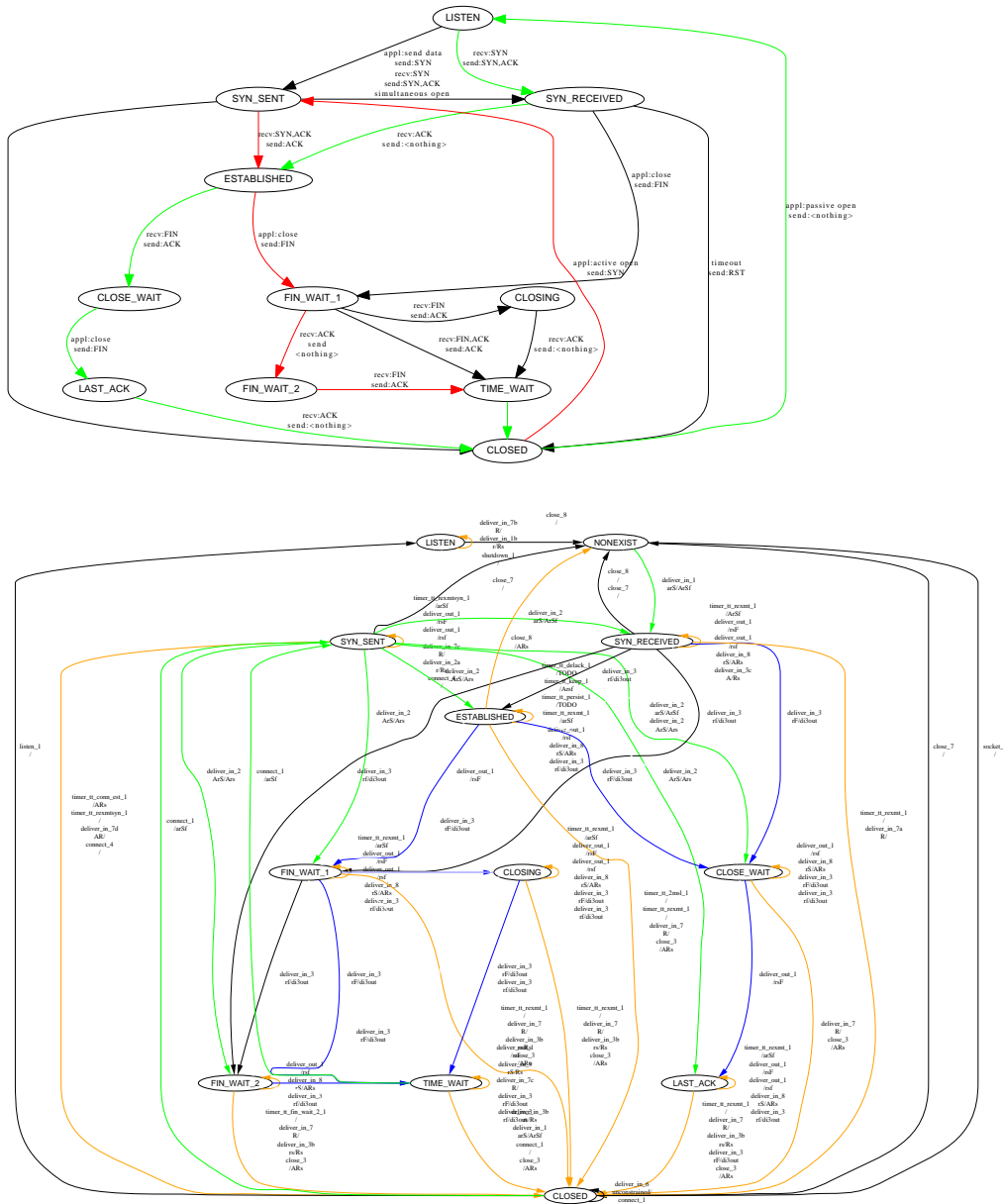
However, with our rigorous protocol-level specification in hand, we can see that, even for the `tcpstate` behaviour, the classic RFC793 and Stevens diagrams give only some of the more common transitions. To show this, we constructed a manual abstract interpretation of our specification, projecting it onto the `tcpstate` component; it is shown at the bottom of Fig. 17 (Page 58)

Comparing the two, one can see that there are many more possible transitions in the model. This suggests that such abstractions could be a useful part of protocol design.

It is also clear from working with the specification that several other components of a TCP socket state are as important as the ‘TCP state’ in determining its behaviour: especially the retransmit mode (NONE, REXMTSYN, REXMT, or PERSIST), but also `cantrcvmore`, `cantsndmore`, `tf_needfin`, `sndq` $\neq []$, etc. Obviously simply taking the product of these would yield an undrawable diagram, but reclustered (slicing) in a different way might be useful. For example, most of the TCP code is independent of which state in {ESTABLISHED, CLOSE_WAIT, LAST_ACK, FIN_WAIT_1, CLOSING, FIN_WAIT_2, TIME_WAIT} is current; instead, the retransmit mode is of much more interest. It is possible that coalescing this class, and then taking the product with the retransmit state, would yield a manageable set of nodes. One could also think of high-performance runtime validation that a TCP implementation is within such a transition system.

10. RELATED WORK

There is a vast literature devoted to verification techniques for protocols, with both proof-based and model-checking approaches, e.g. in conferences such as CAV, CONCUR, FORTE, ICNP, SPIN, and TACAS. To the best of our knowledge, however, no previous work approaches a specification dealing with the full scale and complexity of a real-world TCP. In retrospect this is unsurprising: we have depended on automated reasoning tools and on raw compute resources that were simply unavailable in the 1980s or early 1990s.



The traditional diagrams have transitions involving two different sockets, while ours refers strictly to the state of a single socket). The transitions of the lower diagram are a conservative over-approximation to the set of all the transitions in the model which either (1) affect the ‘TCP state’ of a socket, or (2) involve processing TCP segments from the host’s input queue or adding them to its output queue. Transitions involving ICMPs are omitted, as are those modelling the pathological BSD behaviour in which arbitrary sockets can be moved to LISTEN. Transitions are labelled by their Host LTS rule name (e.g. *socket_1*, *deliver_in_3*, etc.), any socket call involved (e.g. *close()*), and constraints on the ACK/RST/SYN/FIN flags of any TCP segment received and sent, with *a/r/s/f* indicating the flag is clear and *A/R/S/F* indicating it is set. For segments sent by *deliver_in_3*, the flag constraints are simply marked *d3out*. Transitions involving segments (either inbound or outbound) with RST are coloured orange; others with SYN are coloured green; others with FIN are coloured blue; others are coloured black. The FIN indication includes the case of FINs that are constructed by reassembly rather than appearing in a literal segment.

Fig. 17. The TCP state diagram, following Stevens and abstracted from the specification

The most detailed rigorous specification of a TCP-like protocol we are aware of is that of Smith [Smith 1996], 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 [Smith and Ramakrishnan 2002].

Musuvathi and Engler have applied their CMC model-checker to a Linux TCP/IP stack [Musuvathi and Engler 2004]. Interestingly, they began by trying to work with just the TCP-specific part of the codebase (c.f. the pure transport-protocol specification mentioned in §??), but moved to working with the entire codebase on finding the TCP – IP interface too complex. The properties checked were of two kinds: resource leaks and invalid memory accesses, and protocol-specific properties. The latter were 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, with flow control, congestion control etc. not included. Four bugs in the Linux implementation were found.

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 [Bhargavan et al. 2002], 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 naive line count) around 50 times smaller than the specification we present here.

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 [Bhargavan et al. 2001]. They show that certain properties of TCP implementations can be expressed. Lee et al conduct passive testing of an OSPF implementation against an extended finite state machine model [Lee et al. 2002].

There are I/O automata specifications and proof-based verification for aspects of the Ensemble group communication system by Hickey, Lynch, and van Renesse [Hickey et al. 1999], and NuPRL proofs of fast-path optimizations for local Ensemble code by Kreitz [Kreitz 2004].

Alur and Wang address the PPP and DHCP protocols, for each checking refinements between models that are manually extracted from the RFC specification and from an implementation [Alur and Wang 2001].

For radically idealised variants of TCP, one has for example the PVS verification of an improved Sliding Window protocol by Chklyev et al [Chklyev et al. 2003], and Fersman and Jonsson’s application of the SPIN model checker to a simplified version of the TCP establishment/teardown handshakes [Fersman and Jonsson 2000]. Schieferdecker verifies a property (expressed in the modal μ calculus) of a LOTOS specification of TCP, showing that data is not received before it is sent [Schieferdecker 1996]. The specification is again roughly at the level of the TCP state diagram. Billington and Han have produced a coloured Petri net model of the service provided by TCP (in our terminology, roughly an end-to-end specification), but for a highly idealised ISO-style interface, and a highly idealised model of transmission for a bounded-size medium [Billington and Han 2003; Billington and Han 2004]. Murphy and Shankar verify some safety properties of a 3-way handshake protocol analogous to that in TCP [Murphy and Shankar 1987] and of a transport protocol based on this [Murphy and Shankar 1988]. Finally, Postel’s PhD thesis gave protocol models for TCP precursors in a modified Petri net style [Postel 1974].

Implementations of TCP in high-level languages have been written by Biagioni in Standard ML [Biagioni 1994], by Castelluccia et al in Esterel [Castelluccia et al. 1997], and by Kohler et al in Prolac [Kohler et al. 1999]. Each of these develops compilation techniques for performance. They are presumably more readable than low-level C code, but each is a particular implementation rather than a specification of a range of allowable behaviours: as for any implementation nondeterminism means they could not be used as oracles for system testing. Hofmann and Lemmen report on testing of a protocol stack generated from an SDL specification of TCP/IP [Hofmann and Lemmen 2000].

Few details of the specification are given, though it is said to be based on RFCs 793 and 1122. The focus is on performance improvement of the resulting code.

A number of tools exist for testing or fingerprinting of TCP implementations with hand-crafted ad-hoc tests, not based on a rigorous specification. They include the `tcpanaly` of Paxson [Paxson 1997], the TBIT of Padhye and Floyd [Padhye and Floyd 2001], and Fyodor's `nmap` [Fyodor]. RFC2398 [Parker and Schmechel 1998] lists several other tools. There are also commercial products such as Ixia's Automated Network Validation Library (ANVL) [IXIA 2005], with 160 test cases for core TCP, 48 for Slow Start, Congestion Control, etc., and 48 for High Performance and SACK extensions.

11. DISCUSSION

At the start of this project, around October 2000, our aim was twofold:

- (1) to bring mathematical rigour to bear on real-world network protocols, to make it possible to reason formally about them and about distributed systems above them; and
- (2) to bring added realism to the modelling of concurrency, communication, and failure, which were often studied in highly idealised forms in the process calculus and concurrency theory communities.

We were inspired in part by the work on x86 Typed Assembly Language of Morrisett et al. [Morrisett et al. 1999], combining machine realism and rigour to good effect.

As we gained more experience in the area, and with the benefit of hindsight, our focus has shifted, from the desire to support formal correctness reasoning to the general problem we described in §1: the limitations of standard industry engineering practice, with its testing-based development using specifications that are only prose documents, and hence which cannot directly be used in testing.

Our techniques can be described as engineering with logic: 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, and for which formal proof about their behaviour is not yet feasible.

The work has produced contributions of many kinds:

- clear articulation of the notion of specifications that are executable as test oracles, and the benefits thereof;
- demonstration that it is feasible to develop a rigorous specification, in mechanised mathematics, that is executable as a test oracle for an existing real-world key abstraction: the TCP and UDP protocols and their Sockets API;
- the ideas and tools that made it possible to do that:
 - our experimental semantics development process;
 - a clear understanding of the importance of nondeterminism, in the forms of loose specification and implementation runtime nondeterminism, and especially internal nondeterminism, in test-oracle specifications and in their validation;
 - a clear understanding of how the relationship between real system and specification has to be reified in the model and in the testing infrastructure;
 - the specification idioms we needed;
 - the trace-checking technology we developed; and
 - the test generation and instrumentation we developed;
- our specifications themselves: the protocol-level and service-level specifications and the abstraction function between them;
- the idea of experimentally validating such an abstraction function; and
- our demonstration of the discriminating power of our validation process, which found many obscure anomalies and implementation differences.

The total effort required for the project has been around 11 person-years, of which much was devoted to idiom and tool development. This is substantial, and might not be well-motivated for software that is not in critical use, but it is small compared with the effort devoted over the last 30 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.

In the remainder of this final section we reflect on our experience, discussing how the specification might be used and built on, and on the lessons we can draw from this work for design and specification in general, especially for other network protocols.

11.1. Using the Specifications

Our specifications may be of use in several different ways, by several different communities.

11.1.1. As an Informal Reference. Most directly, the annotated and typeset versions of our specifications [Bishop et al. 2005b; Bishop et al. 2005c; Ridge et al. 2009] can be used informally as a reference, as documentation for users and implementors of TCP and the sockets API (designers of distributed systems and implementors of protocol stacks respectively).

A possible objection to this is the unfamiliarity of most engineers with the higher-order-logic language that our specifications are expressed in. There is indeed a small initial overhead in adapting to the notation, for those who typically work only with C or C++ code, but we do not believe that this is a major problem for anyone motivated to look at the specifications, and it is ameliorated by our textual annotation and typesetting. We have seen practising software engineers, without detailed knowledge of TCP or previous exposure to HOL, use the specification to resolve subtle questions about TCP behaviour.

11.1.2. For Bug-fixing. Finding bugs was not one of our goals, but our work identified 33 issues with the implementations we tested, detailed in a technical report [Bishop et al. 2005b, §9], and 260+ places in our specifications where those three implementations differ. Most of the former appear to us to be errors, many of which would be very hard to find with normal testing, which may be worth considering by the current maintainers of these implementations. The nature of those issues shows how discriminating testing against our specifications can be.

11.1.3. As a Basis for New Implementations.

11.1.4. For Testing New Implementations. We developed our specifications in large part by reverse-engineering from three specific implementations (together with careful reading of the existing RFCs and texts), building validation tools to test the specifications against existing implementations. But the same tools could be used to test future implementations against the specifications, for high-quality automated conformance testing. Doing so would be mainly a matter of engineering, to package up our testing tools to make them usable in a more push-button fashion. We would have liked to do this, but lacked the resource required (perhaps a person-year of effort). Having done so, one could test a wide variety of implementations and track implementations through version changes, evolving the specification as required (whereas here we froze the three implementations we considered at the start of our work on TCP).

Interpreting the output of our trace-checking tools is a moderately skilled task that involves diagnosing any discrepancy between implementation behaviour and what the specifications allow, tracking it down to either a bug in the implementation, an error in the specifications, or a limitation of our symbolic evaluation tools. Updating the specification as appropriate requires some familiarity with the HOL language but does not need real theorem-proving expertise (one member of the project, then an undergraduate intern, went from zero knowledge of HOL to contributing useful specification in a matter of weeks).

The performance and maintenance of our symbolic evaluation tools are potential issues, however. As described earlier, they were fast enough to be usable, but still a major bottleneck. Changing the

symbolic evaluator (which might be required by any significant change in the specification) is a highly skilled task, needing deep familiarity with HOL4.

11.1.5. As a Reference for Proposed Changes. TCP continues to evolve, albeit slowly, and our specifications could provide a basis for precise discussion of future changes. It would be desirable to isolate the aspects of the specification that are subject to more frequent change, e.g. restructuring to factor the congestion control aspects out into replaceable modules.

11.1.6. As a New Internet Standard. In principle, our specifications could form the basis for new IETF Internet Standards for TCP, UDP and Sockets, much more rigorous and detailed than the existing RFCs. This is more a political question than a technical one, requiring engagement with the IETF community that we did not embark on (though one would want to do further testing, as outlined above, to ensure that the specifications are not over-fitted to the three implementations we tested).

11.1.7. As a Basis for Formal Reasoning. Given our protocol-level and service-level specifications, one could think of formal mechanised proof:

- (1) correctness of the executable code of a TCP implementation with respect to the protocol-level specification (either one of the existing mainstream C implementations or of a clean reimplementation written with verification in mind); or
- (2) correctness of our protocol-level specification with respect to our service-level specification.

Together these would “prove TCP correct”, establishing much greater confidence in the relationships that we established only by testing here. However, the scale and complexity of the specifications, and of any TCP implementation, make either an intimidating prospect with current proof technology.

One could also attempt to prove correctness of higher-level communication abstractions implemented above our service-level specification, as Compton [Compton 2005] and Ridge [Ridge 2009] did above our UDP specification and a simplified TCP specification. This would be verification of executable implementations rather than the more usual distributed-algorithm proofs about non-executable pseudocode or automata-theoretic descriptions.

11.2. General Lessons for New System and Protocol Development

The experience of this work influenced our later work on semantics for other aspects of real-world systems, including multiprocessor concurrency [Sewell et al. 2010; Sarkar et al. 2011; Flur et al. 2016], C/C++11 concurrency [Batty et al. 2011b], the C language [Memarian et al. 2016], ELF linking [Kell et al. 2016], POSIX filesystems [Ridge et al. 2015], and the TLS protocol [Kaloper-Mersinjak et al. 2015]. All of these involve specifications that are in some way executable as test oracles, and many involved investigation of de facto standards that hitherto have only been expressed as prose specification documents.

We have demonstrated that pre-existing real-world protocols can be specified rigorously, but at the cost of significant effort and using relatively sophisticated tools. We now reflect on why this was necessary for TCP/IP/Sockets, and argue that applying rigorous methods at design-time could be done much more straightforwardly, and have even greater benefits.

Our first lesson is that:

Any specification should be testable.

Industrial software development relies on testing, in a code-test-and-debug cycle, but prose specifications like the TCP/IP RFCs cannot be used directly in that process. At the very least, a good specification should provide a decision procedure for testing (a test oracle): it should define the interface of the system being specified, e.g. here the Sockets API and wire interface events, and, given

an observed trace of those events from an implementation execution, it should let one compute whether that trace is legal or not.

This brings us to our second point, which is that:

Managing specification looseness is key.

In the simplest case, one has a tight specification that is completely deterministic, allowing no freedom for variation in the observable behaviour (except performance) of conforming implementations. Such a specification can trivially be used as a test oracle, simply by running it and an implementation on the same inputs and comparing the results, and it can also serve directly as an executable prototype implementation. In this case one can write the specification in a conventional programming language — perhaps best in the pure fragment of a functional language (such as OCaml, Haskell, or F[#]) or in a specification language that supports code extraction (such as Coq, Isabelle/HOL, HOL4, or Lem), for maximum clarity and to prevent accidental introduction of behavioural complexity from imperative code.

The situation we have dealt with here is at the opposite extreme: our TCP/IP/Sockets specifications had to be highly nondeterministic (c.f. §?? and §3.3.1) to admit the variation in implementation behaviour, and that meant that we had to use a rich mathematical language to express them (in our case, the higher-order logic of HOL4), and we had to build sophisticated tools (as described in §7) to construct test oracles from the specifications. Looking in more detail at the kinds of loose specification that were required suggests several points to bear in mind for design-time specification:

11.2.1. Implementation-internal Nondeterminism. Some nondeterminism is required to accommodate the run-to-run variation in behaviour of a single implementation, e.g. the variations due to OS thread scheduling, pseudo-random choices, and timer rate variations. We modelled these with internal nondeterminism, and our test oracle dealt with them search over the transition graph and by constraint solving. In retrospect, it would have been better to handle these by instrumenting the implementations to emit visible labels at these points, letting the test oracle resolve these choices immediately (we were overly concerned not to modify the systems that we were analysing, and also did not want to rule out testing of the Windows implementations, for which we did not have source access); this would make checking much simpler and faster. There are now widely-used tracing frameworks, e.g. dtrace (http://elinux.org/Kernel_Trace_Systems) that would simplify the instrumentation required.

11.2.2. Superficial Inter-implementation Variation. In some cases we dealt with inter-implementation variation by parameterising the specification, as for the BSD-specific behaviour in the *bind_5* of Fig. 6 (Page 28). This is not a problem for testing, as one knows when using the test oracle which implementation is being used, but such variations are undesirable (as they might well lead to hard-to-detect portability issues) and should rarely be needed if doing design-time specification rather than capturing existing implementations that have already diverged.

11.2.3. Debatable Looseness. In other cases it is debatable how loose a specification should be. For example, when a TCP endpoint receives overlapping TCP segments, we chose in our specification to permit them to be reassembled into a stream in any reasonable way, as we expect there might well be inter-implementation variation here. This required extra work to make a test oracle, as described in §7.4. It is also arguably poor protocol design, as it raises the possibility that a firewall will reassemble the segments in one way, analyse the result, and pass it through, while the endpoint will reassemble them in another way into a different stream. For a new protocol design, one might want to minimise this kind of looseness, both for checking and for robustness.

11.2.4. Major Variations. Finally, one has major variations, where the protocol specification should be intentionally loose to allow real implementation variation and protocol improvements over time. For TCP the obvious example is the congestion control subsystem of the protocol. Here we would have liked to factor that out into a pluggable part of the specification, restricting the main

specification to just enough to ensure functional correctness of the protocol. If done at design-time, we believe that that would have brought a useful focus on the minimal correctness properties that the protocol was intended to achieve.

Our last main point is that:

Rigorous specification can help manage protocol complexity.

As Anderson *et al.* write in their *Design guidelines for robust Internet protocols* (“Guideline #1: Value conceptual simplicity”, [Anderson et al. 2003]), the value of *simplicity* is widely accepted but hard to realise. Writing a behavioural specification makes complexity apparent, drawing attention to unnecessary irregularities and asymmetries in a way that informal prose and working code do not. One is naturally led to consider each case, e.g. of an arbitrary TCP segment arriving at a host in an arbitrary state, whereas when working with a prose specification it is all too easy to add a paragraph without thinking of all the consequences, and when working with code it is all too easy to not consider some code-flow path. Doing this at design-time would help keep the design as simple and robust as possible. It also opens the possibility of machine-checking completeness: that the specification does indeed handle every case (for an earlier version of our specification, we proved a receptiveness property along those lines).

Specification is a form of communication, both within the protocol design group and later to implementors and users. The added *clarity* of rigorous specification aids precise communication and reduces ambiguity.

Drawing these together, if we were to design a new network protocol, we would:

- (1) Clearly identify the part of the overall system that the specification is intended to cover (§??), defining the observable events that it should be in terms of, and how they would relate to actual implementation events. Probably this should include both wire and software API interfaces.
- (2) Specify both the service that the protocol is intended to achieve (as for our service-level stream specification of TCP) and the protocol internals (as our protocol-level segment/endpoint specification), and the relationship between the two.
- (3) Take care to minimise and contain any looseness in the specification, and, aided by that, arrange so that an efficient test oracle can be built directly from the specification.
 - (a) In some cases, one could arrange for the specification to be completely deterministic between observable events, and there one could write those parts of the specification in an executable pure functional language, and then use that directly for testing and as an executable prototype.
 - (b) In other cases, where one really does want to leave implementation freedom (e.g. bounding TCP congestion control within some limits) that should be factored out, and one either needs a more expressive specification language (as here) and a constraint-solving checker, or one should write a test oracle directly.
- (4) Either test (or ideally prove) that the protocol-level specification does provide the intended service.
- (5) Set up random test generation infrastructure, tied to the test oracle, to use for implementations.

Our experience in doing this has been very positive: we specified, in collaboration with the designers, a new Media Access Control (MAC) protocol for the SWIFT experimental optically switched network, by Dales and others [Biltcliffe et al. 2006]; extracting a verified checker from a HOL4 specification and using that to check traces from ns2 and hardware simulations. With relatively low effort we quickly established a high degree of confidence in the protocol design and in its implementation, clarifying several aspects of the design in the process.

Acknowledgements We thank Jon Crowcroft and Tim Griffin for helpful comments, many members of the Laboratory, especially Pieter Brooks, for making compute resource available, and Adam Biltcliffe for his work on testing infrastructure.

REFERENCES

- ALUR, R. AND WANG, B.-Y. 2001. Verifying network protocol implementations by symbolic refinement checking. In *Proc. CAV '01, LNCS 2102*. 169–181.
- ANDERSON, T. E., SHENKER, S., STOICA, I., AND WETHERALL, D. 2003. Design guidelines for robust internet protocols. *Computer Communication Review* 33, 1, 125–130.
- BATTY, M., OWENS, S., SARKAR, S., SEWELL, P., AND WEBER, T. 2011a. Mathematizing C++ concurrency. In *Proc. POPL*.
- BATTY, M., OWENS, S., SARKAR, S., SEWELL, P., AND WEBER, T. 2011b. Mathematizing C++ concurrency. In *Proceedings of POPL 2011: the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 55–66.
- BHARGAVAN, K., CHANDRA, S., MCCANN, P. J., AND GUNTER, C. A. 2001. What packets may come: automata for network monitoring. In *Proc. POPL*. 206–219.
- BHARGAVAN, K., OBRADOVIC, D., AND GUNTER, C. A. 2002. Formal verification of standards for distance vector routing protocols. *J. ACM* 49, 4, 538–576.
- BIAGIONI, E. 1994. A structured TCP in Standard ML. In *Proc. SIGCOMM '94*. 36–45.
- BILLINGTON, J. AND HAN, B. 2003. On defining the service provided by TCP. In *Proc. ACSC: 26th Australasian Computer Science Conference*. Adelaide.
- BILLINGTON, J. AND HAN, B. 2004. Closed form expressions for the state space of TCP's data transfer service operating over unbounded channels. In *Proc. ACSC: 27th Australasian Computer Science Conference*. Dunedin, 31–39.
- BILTCLIFFE, A., DALES, M., JANSEN, S., RIDGE, T., AND SEWELL, P. 2006. Rigorous protocol design in practice: An optical packet-switch MAC in HOL. In *Proc. ICNP, The 14th IEEE International Conference on Network Protocols*.
- BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. 2005a. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proc. SIGCOMM 2005 (Philadelphia)*.
- BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. 2005b. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview. Tech. Rep. UCAM-CL-TR-624, Computer Laboratory, University of Cambridge. Mar. 88pp. Available at <http://www.cl.cam.ac.uk/users/pes20/Netsem/>.
- BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. 2005c. TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The specification. Tech. Rep. UCAM-CL-TR-625, Computer Laboratory, University of Cambridge. Mar. 386pp. Available at <http://www.cl.cam.ac.uk/users/pes20/Netsem/>.
- BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. 2006. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *POPL'06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 55–66.
- CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. 2004. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '04. USENIX Association, Berkeley, CA, USA, 2–2.
- CASTELLUCCIA, C., DABBOUS, W., AND O'MALLEY, S. 1997. Generating efficient protocol code from an abstract specification. *IEEE/ACM Trans. Netw.* 5, 4, 514–524. Full version of a paper in SIGCOMM '96.
- CHKLIAEV, D., HOOMAN, J., AND DE VINK, E. 2003. Verification and improvement of the sliding window protocol. In *Proc. TACAS'03, LNCS 2619*. 113–127.
- CLARK, D. 1992. A cloudy crystal ball — visions of the future. In *Proceedings of the Twenty-Fourth Internet Engineering Task Force*. 539–544. <http://www.ietf.org/proceedings/24.pdf>. Retrieved 2012/12/31.
- COMPTON, M. 2005. Stenning's protocol implemented in UDP and verified in Isabelle. In *CATS*, M. D. Atkinson and F. K. H. A. Dehne, Eds. CRPIT Series, vol. 41. Australian Computer Society, 21–30.
- ELLISON, C. AND ROŞU, G. 2012. An executable formal semantics of C with applications. In *Proc. POPL*.
- FERSMAN, E. AND JONSSON, B. 2000. Abstraction of communication channels in Promela: A case study. In *Proc. 7th SPIN Workshop, LNCS 1885*. 187–204.
- FLUR, S., GRAY, K. E., PULTE, C., SARKAR, S., SEZGIN, A., MARANGET, L., DEACON, W., AND SEWELL, P. 2016. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
- FYODOR. nmap. <http://www.insecure.org/nmap/>.
- GORDON, M., MILNER, R., AND WADSWORTH, C. P. 1979. *Edinburgh LCF, LNCS 78*.
- GORDON, M. J. C. AND MELHAM, T., Eds. 1993. *Introduction to HOL: a theorem proving environment*. Cambridge University Press.

- HICKEY, J., LYNCH, N. A., AND VAN RENESSE, R. 1999. Specifications and proofs for Ensemble layers. In *Proc. TACAS, LNCS 1579*. 119–133.
- HOFMANN, R. AND LEMMEN, F. 2000. Specification-driven monitoring of TCP/IP. In *Proc. 8th Euromicro Workshop on Parallel and Distributed Processing*.
- HOL 2005. The HOL 4 system, Kananaskis-3 release. hol.sourceforge.net.
- IEEE. 2000. *Information Technology—Portable Operating System Interface (POSIX)—Part xx: Protocol Independent Interfaces (PII), P1003.1g*. Institute of Electrical and Electronics Engineers.
- IXIA. 2005. IxANVL(tm) — automated network validation library. <http://www.ixiacom.com/>.
- KALOPEL-MERSINJAK, D., MEHNERT, H., MADHAVAPEDDY, A., AND SEWELL, P. 2015. Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation. In *USENIX Security 2015: 24th USENIX Security Symposium, Washington, D.C., USA*. 223–238.
- KELL, S., MULLIGAN, D. P., AND SEWELL, P. 2016. The missing link: explaining ELF static linking, semantically. In *OOPSLA 2016: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA.
- KOHLER, E., KAASHOEK, M. F., AND MONTGOMERY, D. R. 1999. A readable TCP in the Prolac protocol language. In *Proc. SIGCOMM '99*. 3–13.
- KREBBERS, R. 2015. The c standard formalized in coq. Ph.D. thesis, Radboud University Nijmegen.
- KREITZ, C. 2004. Building reliable, high-performance networks with the Nuprl proof development system. *J. Funct. Program.* 14, 1, 21–68.
- LEE, D., CHEN, D., HAO, R., MILLER, R. E., WU, J., AND YIN, X. 2002. A formal approach for passive testing of protocol data portions. In *Proc. ICNP*.
- LEROY, X. ET AL. 2004. *The Objective-Caml System, Release 3.08.2*. INRIA. Available <http://caml.inria.fr/>.
- LI, P. 2008. Programmable concurrency in a pure and lazy language. Ph.D. thesis, University of Pennsylvania.
- LI, P. AND ZDANCEWIC, S. http://www.cl.cam.ac.uk/~pes20/Netsem/unify_with_tcp.tgz.
- LI, P. AND ZDANCEWIC, S. 2007. Combining events and threads for scalable network services. In *Proc. PLDI*. 189–199.
- LYNCH, N. AND VAANDRAGER, F. 1996. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation* 128, 1, 1–25.
- MCKUSICK, M. K., NEVILLE-NEIL, G., AND WATSON, R. N. 2014. *The Design and Implementation of the FreeBSD Operating System* 2nd Ed. Addison-Wesley Professional.
- MEMARIAN, K., MATTHIESEN, J., LINGARD, J., NIENHUIS, K., CHISNALL, D., WATSON, R. N., AND SEWELL, P. 2016. Into the depths of C: elaborating the de facto standards. In *PLDI 2016: 37th annual ACM SIGPLAN conference on Programming Language Design and Implementation (Santa Barbara)*.
- MORRISSETT, G., CRARY, K., GLEW, N., GROSSMAN, D., SAMUELS, R., SMITH, F., WALKER, D., WEIRICH, S., AND ZDANCEWIC, S. 1999. Talx86: A realistic typed assembly language. In *In Second Workshop on Compiler Support for System Software*. 25–35.
- MURPHY, S. L. AND SHANKAR, A. U. 1987. A verified connection management protocol for the transport layer. In *Proc. SIGCOMM*. 110–125.
- MURPHY, S. L. AND SHANKAR, A. U. 1988. Service specification and protocol construction for the transport layer. In *Proc. SIGCOMM*. 88–97.
- MUSUVATHI, M. AND ENGLER, D. 2004. Model checking large network protocol implementations. In *Proc. NSDI: 1st Symposium on Networked Systems Design and Implementation*. 155–168.
- Netsem github 2015. Netsem github repository. <https://github.com/PeterSewell/netsem>.
- NORRISH, M. 1998. C formalised in HOL. Tech. Rep. UCAM-CL-TR-453, U. Cambridge, Computer Laboratory.
- NORRISH, M., SEWELL, P., AND WANSBROUGH, K. 2002. 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)*. 49–53.
- PADHYE, J. AND FLOYD, S. 2001. On inferring TCP behaviour. In *Proc. SIGCOMM '01*.
- PAPASPYROU, N. S. 1998. A formal semantics for the C programming language. Ph.D. thesis, National Technical University of Athens.
- PARKER, S. AND SCHMECHEL, C. 1998. RFC2398: Some testing tools for TCP implementors.
- PAXSON, V. 1997. Automated packet trace analysis of TCP implementations. In *Proc. SIGCOMM '97*. 167–179.
- POSTEL, J. 1974. *A Graph Model Analysis of Computer Communications Protocols*. University of California, Computer Science Department, PhD Thesis.
- RIDGE, T. 2009. Verifying distributed systems: the operational approach. In *Proc. POPL*. 429–440.

- RIDGE, T., NORRISH, M., AND SEWELL, P. 2008. A rigorous approach to networking: TCP, from implementation to protocol to service. In *Proc. FM'08: 15th International Symposium on Formal Methods (Turku, Finland)*, LNCS 5014. 294–309.
- RIDGE, T., NORRISH, M., AND SEWELL, P. 2009. TCP, UDP, and Sockets: Volume 3: The service-level specification. Tech. Rep. UCAM-CL-TR-742, Computer Laboratory, University of Cambridge. Feb. 305pp. Available at <http://www.cl.cam.ac.uk/users/pes20/Netsem/>.
- RIDGE, T., SHEETS, D., TUERK, T., GIUGLIANO, A., MADHAVAPEDDY, A., AND SEWELL, P. 2015. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *SOSP 2015: Proceedings of the 25th Symposium on Operating Systems Principles, Monterey, CA, USA*. 38–53.
- SARKAR, S., SEWELL, P., ALGLAVE, J., MARANGET, L., AND WILLIAMS, D. 2011. Understanding POWER multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*. 175–186.
- SCHIEFERDECKER, I. 1996. Abruptly-terminated connections in TCP – a verification example. In *Proc. COST 247 International Workshop on Applied Formal Methods In System Design*.
- SERJANTOV, A., SEWELL, P., AND WANSBROUGH, K. 2001a. The UDP calculus: Rigorous semantics for real networking. Tech. Rep. 515, Computer Laboratory, University of Cambridge. July.
- SERJANTOV, A., SEWELL, P., AND WANSBROUGH, K. 2001b. The UDP calculus: Rigorous semantics for real networking. In *Proc. TACS 2001: Fourth International Symposium on Theoretical Aspects of Computer Software, Tohoku University, Sendai*.
- SEWELL, P., SARKAR, S., OWENS, S., ZAPPA NARDELLI, F., AND MYREEN, M. O. 2010. x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors. *Communications of the ACM* 53, 7, 89–97. (Research Highlights).
- SMITH, M. A. AND RAMAKRISHNAN, K. K. 2002. Formal specification and verification of safety and performance of TCP selective acknowledgment. *IEEE/ACM Trans. Netw.* 10, 2, 193–207.
- SMITH, M. A. S. 1996. Formal verification of communication protocols. In *Proc. FORTE 96. Formal Description Techniques IX: Theory, application and tools, IFIP TC6 WG6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI*. 129–144.
- STEVENS, W. R. 1994. *TCP/IP Illustrated Vol. 1: The Protocols*.
- STEVENS, W. R. 1998. *UNIX Network Programming Vol. 1: Networking APIs: Sockets and XTI* Second Ed.
- THE NETSEM PROJECT. Web page. <http://www.cl.cam.ac.uk/users/pes20/Netsem/>.
- WANSBROUGH, K., NORRISH, M., SEWELL, P., AND SERJANTOV, A. 2002. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proc. ESOP, LNCS 2305*. 278–294.
- WRIGHT, G. R. AND STEVENS, W. R. 1995. *TCP/IP Illustrated Vol. 2: The Implementation*.
- YI, W. 1991. CCS + time = an interleaving model for real time systems. In *Proc. ICALP*. 217–228.