Rigorous Protocol Design in Practice: An Optical Packet-Switch MAC in HOL

Adam Biltcliffe*, Michael Dales†, Sam Jansen†, Thomas Ridge*, Peter Sewell*

*Computer Laboratory
University of Cambridge
First.Last@cl.cam.ac.uk

†Intel Research
Cambridge
{Michael.W.Dales,Sam.Jansen}@intel.com

***Abstract*— This paper reports on an experiment in network protocol design: we use novel rigorous techniques in the design process of a new protocol, in a close collaboration between systems and theory researchers.**

The protocol is a Media Access Control (MAC) protocol for the SWIFT optical network, which uses optical switching and wavelength striping to provide very high bandwidth packet-switched interconnects. The use of optical switching (and the lack of optical buffering) means that the protocol must control the switch within hard timing constraints.

We use higher-order logic to express the protocol design, in the general-purpose HOL automated proof assistant. The specification is thus completely precise, but still concise, readable, and without accidental overspecification. Further, we test conformance between the specification and two implementations of the protocol: an NS-2 simulation model and the VHDL code of the network hardware. This involves: (1) proving, within HOL, that the specification is equivalent to an algorithmically-checkable version; (2) using automatic code-extraction to generate a testing oracle; and (3) applying that oracle to traces of the implementation.

This design-time use of rigorous methods has resulted in a protocol that is better specified and more correct than it would otherwise be, with relatively little effort.

I. Introduction

Network protocols are often designed using informal prose specifications of the intended behaviour, as found in most RFCs, coupled with experimental implementation code. This dependence on prose specifications has several problems:

- they are imprecise, so there may be misunderstandings among designers and implementors as to exactly what the specification means;
- there is no way to test conformance of an implementation directly against the specification, so implementation differences tend to proliferate unchecked; and
- they make it hard to see the protocol as a whole, to identify any unnecessary complexity or unexpected interactions between features.

In response to these problems, there has been a great deal of work on mathematically rigorous techniques for expressing protocol behaviour, and on various kinds of formal verification. Even now, though, most formal work deals with highly idealised protocols, and most practical protocol design is unsupported by rigorous descriptions of endpoint behaviour.

In previous research we established specification and automated testing techniques that are effective for complex real-world protocols, with a detailed and accurate model of TCP, UDP, and the Sockets API [5], [6]. That work was post hoc, describing the behaviour of existing protocols. In contrast, this paper reports on an experiment in rigorous protocol *design*, and the related specification and testing techniques, in an ongoing systems project. The benefits of rigorous behavioural description at design-time are potentially much greater than for post hoc work, as infelicities in the protocol can be identified and fixed very early, but it requires an agile process. We describe how we have achieved this, with a close collaboration between systems and theory researchers.

The Protocol The protocol we address is a Media Access Control (MAC) protocol for a high-capacity optically-switched packet-switched network: the SWIFT network [12], [16]. It is described informally in §II.

The SWIFT network seeks to meet ongoing high-capacity interconnect requirements by making the end-to-end data-path all optical, replacing electronic switching components with optical ones. At very high data rates the electronics required to switch high speed data will become increasingly expensive and power hungry. Optical switches remove the need to process the data on the network electronically, providing both a performance and power advantage. However, optical switches fundamentally alter how the network is managed. There is no optical equivalent of RAM, so buffering in switches can no longer be relied upon to resolve contention for output ports. Instead, in order to guarantee delivery of frames, an end-to-end light path must be constructed before the frame is injected into the network. This, coupled with the inability to process packet headers in the traditional manner due to the limits of optical header reading, lead to new network structure requirements for optically switched networks [7].

These changed constraints mean that more complex MAC protocols are needed to manage such interconnects. In addition to typical start-of-day and management tasks, the network control protocol now has to explicitly manage setting up the network for delivery of frames from one host to another — no longer is it sufficient to fire a frame into the network immediately and rely on the network to simply deliver it correctly.

The protocol is a good target for an experiment in design: it involves new and subtle properties and is moderately complex, but is not as large as (say) TCP.

RIGOROUS DESIGN TECHNIQUES Our design approach is explained in §III. It has two main components: a mathematically rigorous style for defining the protocol, and a semi-automated technique for testing conformance between this definition and implementations. For this to be practical, the specification style has to be:

- clear, concise, and readable by all those involved, as its primary use is to facilitate communication among the designers;
- expressive enough to describe the key aspects of the protocol behaviour, especially the timing constraints, without leading the designers to *overspecify* aspects of behaviour which they wish to leave open; and
- machine-processed, as pen-and-paper mathematics on this scale becomes hard to keep consistent (especially for an evolving protocol design), and to support conformance testing.

Higher-order logic and HOL We use higher-order logic to express the specification, mechanised using the HOL automated proof assistant [14]. Higher-order logic is similar to conventional first-order logic, with the normal connectives and quantifiers, etc., but with the addition of a rich type structure (functions, lists, sets, numeric types), and the ability to quantify over any of these types. It is extremely expressive, allowing one to write more-or-less arbitrary mathematics idiomatically.

HOL is a system for manipulating higher-order logic definitions, type-checking them, and performing proof. Large libraries of mathematics have already been developed in it. Automatic type-checking is invaluable, quickly detecting many simple errors, and automatic typesetting prevents transcription errors. The system provides the user with a wide variety of decision procedures and tactics, which assist the process of proof construction. HOL is not a fully automatic theorem prover, or model checker, since the expressiveness of higher-order logic means that sophisticated proofs are hard to discover automatically. However, various fragments, such as first order logic, are fully automatic. Machine-processed mathematics in a system such as HOL, in a well-defined logic, is currently the most rigorous form of definition possible.

The use of higher-order logic may be unfamiliar at first sight, potentially making the specification less accessible. In practice, however, our experience is that this is not a problem. It is important to have some members of the design team fluent in its use, but the more systems-oriented researchers have quickly become sufficiently familiar with it to be able to discuss their intended design in terms of the formalised properties. Thus, whilst less accessible than informal English, the use of formal logic is not the barrier to accessibility that it might appear, whilst it brings with it great benefits in terms of rigour.

Conformance testing by proof and code extraction The specification is written to be as clear as possible, and consequently is not algorithmically checkable. For conformance testing, therefore, we prove, within HOL, that the specification is equivalent to a conformance-checking algorithm. We then use the automatic code-extraction facilities of HOL [14] to generate a testing oracle: an ML program that reads a trace of events and checks whether or not it meets the specification. This is described in §V.

We have applied this to two implementations of the protocol: an NS-2 simulation model and the VHDL code of the network hardware. In each case we generate traces from the implementation (and for the first aiming for as much coverage as possible), and then use the oracle to check them.

Process The whole process is flexible and lightweight. The informal descriptions of the protocol, the formal specification, and the initial implementations complement each other; they have been developed hand-in-hand (we are *not* advocating writing a specification in isolation and then handing it over to implementors). Simply writing the formal specification clarified many important issues in the protocol, and (as we discuss in §VI and §VIII) conformance testing has identified errors both in the specification and in the implementation. Changing the specification is straightforward (though re-proving the algorithmic characterisation of the specification does require some work), and re-running the conformance tests when the implementation changes is automated and reasonably fast.

SUMMARY OF CONTRIBUTION

- We describe a successful experiment in design-time use of rigorous techniques, with a lightweight design-for-test approach, for a novel real-world protocol.
- We construct a testing oracle directly from the specification, using a combination of mechanised proof (within HOL) and automatic extraction of code from the resulting algorithmic specification. This provides very high confidence in the correctness of the oracle.
- Our approach exposed a variety of errors during the design process, in both specification and implementation, both during discussion about the specification and from use of the test oracle on NS-2 and VHDL implementations. We give illustrative examples.

II. SWIFT OPTICAL NETWORK OVERVIEW

There is increasing demand for high-bandwidth, short-scale interconnects, as currently exemplified by Infiniband, Fibre Channel, Gigabit and 10 Gigabit Ethernet, and PCI-Express. Whilst these networks are currently heading into the tens of gigabits, to reach hundreds of gigabits or terabits it is likely that such interconnects will need to turn to optical technologies [10], [13]. In addition to the obvious route of using optical fibre links, with the high capacity that multiplexing several channels over a single fibre can provide, such networks will benefit also from optical switching, in both performance and power.

GENERAL OVERVIEW The SWIFT network architecture is designed to provide a short-range, end-to-end optical, packet-switched, high-bandwidth interconnect. Potential applications include intra- and inter-server backplanes and small LANs. This context imposes a different set of requirements from that of conventional (i.e., long-haul) optical network design. In addition to the network design goals SWIFT has been designed such that it can be constructed using close-to-available technology, and with an eye to keeping the network's cost down. An interconnect that is aimed at potential deployment on motherboards must be relatively cheap.

A SWIFT interconnect is based around a single switch, which is suitable for the limited domains SWIFT is aimed at. We use electrically driven optical devices called Semiconductor Optical Amplifiers (SOAs). By chaining these devices together an optical switch fabric can be constructed. We have successfully demonstrated an optical packet switch constructed from SOAs under electronic control working at 100 Gbps [11].

To achieve these levels of high capacity the SWIFT network uses Wavelength Division Multiplexing (WDM) to send data over multiple wavelengths simultaneously. Typically, in long-haul networks, WDM is used to carry multiple independent channels of data, but in SWIFT we use a technique called wavelength striping, where all wavelengths are used for a single channel of data. This allows us to scale the network easily by adding more wavelengths in addition to the typical approach of adding faster transceivers. Because all wavelengths are switched simultaneously, this simplifies the switch fabric design, removing the need for wavelength sensitive devices.

To perform contention resolution *before* packets enter the network, as there is no optical buffering, hosts connected to a switch use a reserved wavelength on the fibre as a control channel between the host and the switch. This control channel is used by a host to request access to the network for a unit of transmission, and only when an end-to-end light-path has been constructed for this packet does the switch tell the host that it is free to transmit. This is similar to Optical Burst Switching (OBS) which has been proposed for long-haul networks [17]. However, because of the larger network diameter OBS does not wait for a clear to transmit signal, so cannot guarantee delivery — the smaller scale of the SWIFT network allows us to achieve this.

An overview of the SWIFT architecture can be seen in Fig. 1. The network consists of a number of hosts connected to a single switch. Hosts stripe data packets over $n - 1$ wavelengths in the network interface, with the remaining wavelength for point-to-point control channels between the arbiter and each host. All the data wavelengths are simultaneously optically switched to the appropriate output port on the switch. The switch fabric is managed by the arbiter, which accepts transmission requests from hosts over the respective control channels and will set up the switch fabric appropriately.

The switch fabric is configured for a fixed duration called a *slot*, which is long enough to allow transmission of a maximum sized packet. Many light-paths may be valid through the switch fabric in a given slot. When the arbiter schedules light-paths to allow hosts to communicate it does so on a slot by slot basis. Note there is no global clock or notion of a slot: only the arbiter works in terms of slots; the hosts know only to transmit when they are granted permission. For each new slot, the switch fabric is reconfigured and all the hosts than can now transmit during this configuration are sent appropriate grant messages telling them who they can send to. The policy used for scheduling slots can be altered to suit the network usage pattern, ranging from simple round robin scheduling, a priority based scheduler, or a scheduler that attempts to deliver quality of service guarantees.

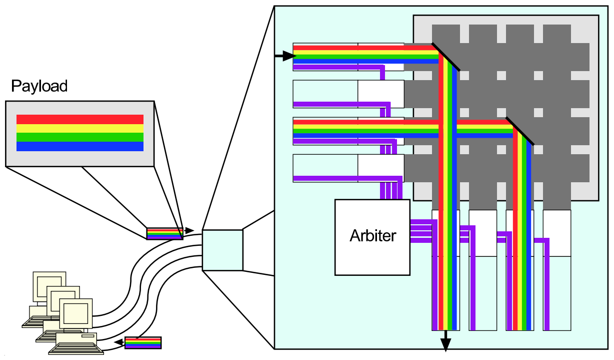


Fig. 1. Overview of the SWIFT network

The arbiter is also responsible for the configuration management of the network. To allow for the most efficient use of the network possible the switch will measure the Round Trip Time between itself and each host, and offset grant messages appropriately. It also carries out basic network configuration tasks such as assigning MAC addresses to hosts.

THE SWIFT MAC PROTOCOL The MAC protocol operates on the control channel and controls access to the data plane. There are three basic processes involved: ping/pong messages, used to assess network liveness and measure link delays; start-of-day messages, used to carry out network configuration such as link layer address assignment; and basic operation messages, which allow hosts to deliver packets to other hosts.

The arbiter sends periodic ping messages out on all ports continuously, with the periodic interval changing depending on whether the link is deemed active or inactive. Initially the link is assumed to be inactive; once a response to the ping (a pong) is received, the arbiter knows the port to be active. The ping interval is smaller when a link is inactive to ensure a fast response time when a host is connected to the network, and then the frequency of ping messages is reduced so as not to clog the control channel while ensuring the host is still alive. Ping and pong messages include an identifier used when calculating Round Trip Time (RTT): the RTT between arbiter and host over the control channel is the time between ping and pong packets with the same identifier on a port — this allows the host to offset time-sensitive messages suitably to take into account propagation delays. Ports are marked as inactive when the fibre goes dark or a host has not responded

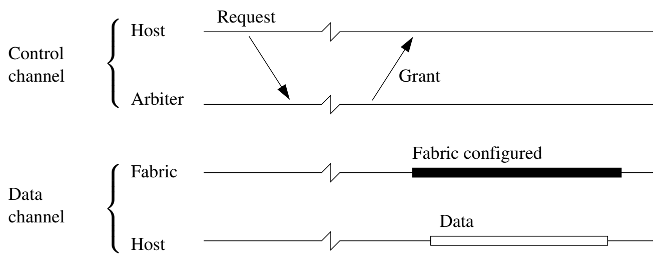


Fig. 2. Requesting a slot and sending data in the SWIFT network

to ping messages for a period.

Once a host has received a ping message and replied, it requests a MAC address from the arbiter. The arbiter responds with a MAC address assignment packet. Once the host is assigned a MAC address it can request to send data to other MAC addresses. A host may also request the MAC address of the *address master*, a host that is assigned to answer Address Resolution Protocol (ARP) look-ups. ARP for the SWIFT network is not covered here, as it not part of the MAC protocol.

When a host is instructed to send a packet, the packet is buffered and a Data Request message containing a destination MAC address is sent to the arbiter. The arbiter will schedule that request into a future slot, and at some point reply with a Data Grant message just prior to that slot (offset by one RTT) to indicate that the switch fabric has been appropriately configured to create a light-path between the source and destination, and the host may now transmit the packet. Data Request messages may be made at any time by the host, asynchronous of any activity on the data plane. However, Data Grant messages are sent just prior to the switch fabric configuration for the appropriate light-path.

Fig. 2 shows the timing relationship between the activities on the control channel and the data plane. Initially the host sends a request message saying it wants to send a frame, and the arbiter will schedule that request for some time later. Some time before the slot starts the arbiter sends a grant message, such that the switch fabric will be configured correctly when the data sent by the host arrives at the switch.

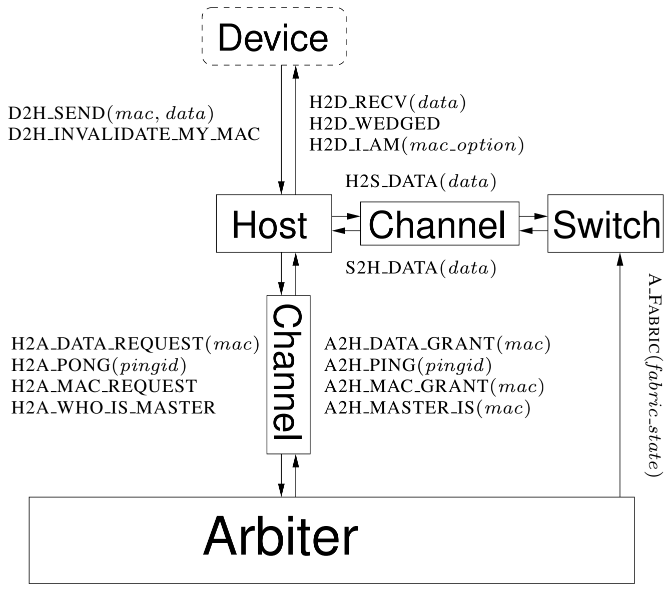
The MAC protocol is designed on the assumption that the control channel is reasonably reliable, with few bit errors. Thus the protocol does not specify acknowledgements for messages, but rather optimises for the common case where messages will be successfully delivered. However, because there will be some possibility of loss, the protocol is designed such that timeouts will be used to force the hosts to resend messages in the event they do not get a reply.

III. Formal, Mechanised Specification

In this section we explain the overall structure of the specification and the idioms used, with a few representative excerpts. The full mechanized specification is around 650 non-comment lines of HOL; it is available online [4].

Choice of observations In writing a specification, we must first choose which events of the system to model, and at what level of abstraction. Our main goal is to capture the behaviour of the components that is required for satisfactory operation of the network. We therefore model the interactions between the hosts, the arbiter, the switch, and the connecting channels, specifying the allowable behaviour of each of these. We also model the interactions between a host and its higher-level device (and device driver, etc.), but do not constrain the behaviour of the device. In particular, we assume some higher-level mechanism which takes care of MAC address resolution.We model individual messages abstractly, omitting low-level details such as their bit-level layout. One could include such details (indeed, earlier versions of the spec did so), but for this protocol they are simple enough that they can be unambiguously specified separately. We therefore focus on the timing and high-level content of messages, for example D2H_SEND$(mac, data)$, for a device-to-host message requesting that $data$ be sent to mac.

The main components of the model are shown below, together with the messages sent between them. Each kind of message is defined as a separate HOL type (e.g. a2h_msg) so that HOL typechecking can catch many simple errors. The diagram shows only one host, but the specification deals with an arbitrary number.



Global trace-based specification The specification is phrased as a predicate on traces of the whole system. Such a trace is a sequence of network labels, each of which is an element of the HOL type below. This is roughly a disjoint union of the messages shown above (some tagged with their port using the HOL pair type constructor #), and three additional cases: N_Dur(time), representing the passage of (real-valued) time; N_Dark(port), indicating that port has gone dark; and N_Tau, a null event.

```
n_lbl =
    N_TAU
```

```
  | N_DUR of time
  | N_FABRIC of fabric_state
  | N_DARK of port
  | N_A2CA of port#a2h_msg#time
  | N_CA2H of port#a2h_msg
  | N_H2CA of port#h2a_msg#time
  | N_CA2A of port#h2a_msg
  | N_D2H of port#d2h_msg
  | N_H2D of port#h2d_msg
  | N_H2CS of port#h2s_msg#time
  | N_CS2S of port#h2s_msg
  | N_S2CS of port#s2h_msg#time
  | N_CS2H of port#s2h_msg
```

For modularity, the specification is phrased as a conjunction of properties: a trace is admissible if its projections onto the host, arbiter and switch components each satisfy the relevant properties. The specification is additionally parameterised on the ports of the arbiter. The top level is therefore as below.

```
spec ports(t : net_trace) =

 arbiter_spec ports(arbiter_trace t) ∧
 switch_spec(switch_trace t) ∧
 (∀p. mem p ports ⟹
   host_spec(host_trace p t) ∧
   arbiter_channel_spec(arbiter_channel_trace p t))
```

Syntax HOL definitions resemble standard mathematical definitions, and include the full range of propositional connectives $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$ as well as the usual quantifiers $\forall x, \exists x$. Application of a function f to an argument x, which is often informally written $f(x)$, is written without brackets in HOL, $f\ x$. A trace t is an infinite sequence of events, expressed as a function from the natural numbers to the set of events. Thus, if t is a trace, the application $t\ n$ gives the event at position n in t. If t is a trace, and P is a property, $P\ t$ expresses that P holds of trace t.

The specification of each component is then expressed simply as a conjunction of properties on traces of events that occur at that component. The arbiter is shown below; the host has a similar definition.

```
arbiter_spec ports(t : arbiter_trace) =

 grants_correctly_arbitered t ∧
 starts_pinging ports t ∧
 continues_pinging t ∧
 pings_correctly_spaced t ∧
 pingids_not_reused_too_soon t ∧
 data_requests_get_granted t ∧
 only_talk_to_ports_with_macs t ∧
 one_mac_per_port t ∧
 mac_requests_are_granted t ∧
 one_port_per_mac t
```

The definitions of these properties are the heart of the spec. Before we examine their formal statements, we give brief informal descriptions of some of the arbiter properties, taken from comments in the spec.

- grants_correctly_arbitered: if the arbiter sends a grant, then the switch must be configured for the associated light path for a certain interval of time.
- starts_pinging: the arbiter must send a ping on all ports before UNCONNECTED_PING_REPEAT_TIME has elapsed.
- continues_pinging: all pings on a port are eventually followed by another ping on the same port.
- pings_correctly_spaced: the interval between consecutive pings on a given port is determined by the arbiter's view of the status of the host on that port.
- pingids_not_reused_too_soon: the arbiter does not reuse pingids within MIN_PING_REUSE_TIME.
- data_requests_get_granted: if the arbiter receives a data-request, it eventually sends a data-grant.
- only_talk_to_ports_with_macs: the arbiter does not send anything (except MAC-grants and pings) to ports with no assigned MAC.
- one_mac_per_port: the arbiter does not send two conflicting MACs to the same port withoutsomething to invalidate the MAC inbetween.
- mac_requests_are_granted: if the arbiter receives a MAC-request, it eventually sends a MAC-grant.
- one_port_per_mac: the arbiter never assigns the same MAC to more than one port.

Let us consider one of the simpler properties, starts_pinging, in more detail.

```
starts_pinging ports(t : arbiter_trace) =

∀p. mem p ports ⟹
∃n pingid.(t n = A_A2H(p, A2H_PING pingid)) ∧
 a_time t n ≤ UNCONNECTED_PING_REPEAT_TIME)
```

Line by line, the property says the following. starts_pinging is a property of an arbiter_trace parameterised by a set of ports (those that are physically connected to the arbiter). If p is a member of the set of ports, then there must exist a point n, and a *pingid* such that $t\ n$ is a ping event with that *pingid*. Moreover, the time at which this event occurs must be less than or equal to UNCONNECTED_PING_REPEAT_TIME.

A more complex property that the arbiter must satisfy is grants_correctly_arbitered, informally described above.

```
grants_correctly_arbitered t =

∀n psrc mac.
(t n = A_A2H(psrc, A2H_DATA_GRANT mac)) ⟹
let rtt_est = a_rtt_estimate t psrc n in
```

$\textbf{case}\ \ rtt\_est\ \ \textbf{of}\ * \rightarrow \textbf{F} \parallel \uparrow rtt \rightarrow$
$mac \in \textbf{dom}((port\_of\_mac\ t\ n)) \wedge$
$\textbf{let}\ \ pdst = (port\_of\_mac\ t\ n)[mac]\ \textbf{in}$
$\textbf{let}\ \ tn = \mathrm{a\_time}\ \ t\ n\ \ \textbf{in}$
$\textbf{let}\ \ low\_time = tn + max(rtt - \mathrm{SLOP\_TIME})0\ \ \textbf{in}$
$\textbf{let}\ \ high\_time =$
$\quad tn + rtt + \mathrm{TRANSMISSION\_TIME} + \mathrm{SLOP\_TIME}\ \ \textbf{in}$
$\exists low\ high.\ \mathrm{a\_time}\ \ t\ low \leq low\_time \wedge$
$high\_time \leq \mathrm{a\_time}\ \ t\ high \wedge$
$\forall n. low \leq n \wedge n \leq high \implies (psrc, pdst) \in\ \mathrm{a\_fabric}\ \ t\ n$

This property states that, if the arbiter sends on port $psrc$ an A2H_DATA_GRANT for destination host identified by mac then the optional rtt estimate rtt_est for $psrc$ cannot be null (written $*$), but must actually be set to some value rtt. If so, it must also be the case that the destination host mac can be identified with a port by the arbiter, using auxiliary function port_of_mac, i.e. mac is in the domain of this function, and $pdst$ is the result of applying the function to mac. We then consider a time interval between low_time and $high\_time$ inclusive. low_time, the start of the interval, is the current time tn plus the rtt minus the SLOP_TIME, except in case SLOP_TIME is bigger than rtt, in which case, low_time is just the current time. $high\_time$ is set similarly. We require two points in the trace, $low, high$ which contain this interval, and such that at every point inbetween, the pair $psrc, pdst$ is a configured path in the switch fabric.

SPECIFICATION STYLE In writing the specification, there were several interesting technical choices of what mathematical idiom to use. These choices were driven by the particular nature of this protocol. For one example, the protocol is asynchronous, without a global clock that is shared between arbiter and hosts, but with real-time constraints on when the switch fabric must be set up. Further, these constraints must allow for varying fibre propagation delays (as they heat and cool). We therefore use traces of instantaneous events interleaved with real-time-passage events, rather than clocked traces with events at fixed intervals. For another example, the state of the arbiter and host is left implicit, with the properties in terms only of observable events. This makes it easier to avoid accidental overspecification and is appropriate for this protocol — in contrast to our earlier work on TCP, where the state structure (TCP control block variables etc.) is needed for the description of congestion control algorithms.

IV. IMPLEMENTATIONS

We have worked with two implementations of the protocol (written by two different authors), described briefly here.

VHDL IMPLEMENTATION A prototype of the SWIFT network has been built at Intel Research, with a three-port optical switch and three hosts. The switch electronics and host electronics have been implemented using Xilinx Virtex-II Pro FPGAs, with the custom hardware required being written in VHDL, a very popular hardware description language. They total over 8600 lines of VHDL.

Ideally conformance testing of the hardware implementation would be carried out at two points. Firstly, during the simulation stage of hardware development, where the VHDL code is executed in a tool such as the popular Modelsim from Mentor Graphics. Hardware simulation is the primary way of debugging hardware before it goes to manufacture. Because hardware mistakes are very expensive to correct, simulation tests need to be as complete as possible. Thus using conformance testing at this stage is ideal. The second point to carry out conformance testing is on the actual hardware when it is built. This allows the designers to confirm that what they have built still conforms to the specification. This stage would require capture of data on the links which could then be turned into a trace.

In this paper we have carried out conformance testing at the first stage. Augmenting the VHDL code with additional debug output to generate a trace suitable for consumption by the checker is relatively straightforward, and can be seen as just an extension of existing debugging of the design. Although it would be ideal to also capture the traces of the live network, the optical nature of the links means that additional splitters in the lines would be necessary. This would affect the optical power budget for the network, and also would require much expensive monitoring. This would be feasible in a commercial environment, but is beyond the time/cost constraints for a research project such as SWIFT.

NS-2 IMPLEMENTATION NS-2 is an object oriented packet-based network simulator written in C++ and OTcl [2]. It includes models for local area networks which define channels, physical interfaces, MAC and link layers. The model for the SWIFT network is built on these base classes with close to 5,000 lines of combined C++ and OTcl. The models implement the specification of the network along with a full address resolution protocol. Simulation scripts written in OTcl are able to modify timing parameters and algorithms used for the various components of the network.

The data and control planes are simulated with two separate sets of network connections, which is logically the same as using different wavelengths. The arbiter can be selected from a set of arbitration policies such as round-robin granting of switch configurations and a simple queue of requests (used to generate the traces tested). It is easy to add tracing functionality to the simulation models; all output is written to a file if tracing is enabled. A short example trace is as follows.

$example\_trace = [\mathrm{N\_DUR}(10 * \mathrm{USEC}),$
$\mathrm{N\_A2CA}(\textbf{n2w}\ 0, \mathrm{A2H\_PING}(\textbf{n2w}\ 901), 84 * \mathrm{NSEC}),$
$\mathrm{N\_DUR}(10 * \mathrm{USEC}),$
$\mathrm{N\_A2CA}(\textbf{n2w}\ 0, \mathrm{A2H\_PING}(\textbf{n2w}\ 902), 84 * \mathrm{NSEC}),$
$\mathrm{N\_DUR}(85 * \mathrm{NSEC}),$
$\mathrm{N\_CA2H}(\textbf{n2w}\ 0, \mathrm{A2H\_PING}(\textbf{n2w}\ 902)),$
$\mathrm{N\_H2CA}(\textbf{n2w}\ 0, \mathrm{H2A\_PONG}(\textbf{n2w}\ 902), 84 * \mathrm{NSEC}),$
$\mathrm{N\_H2CA}(\textbf{n2w}\ 0, \mathrm{H2A\_MAC\_REQUEST}, 84 * \mathrm{NSEC}),$
$\mathrm{N\_DUR}(85 * \mathrm{NSEC}),$
$\mathrm{N\_CA2A}(\textbf{n2w}\ 0, \mathrm{H2A\_PONG}(\textbf{n2w}\ 902)),$
$\mathrm{N\_DUR}(80 * \mathrm{NSEC}),$

$\text{N\_CA2A}(\mathbf{n2w}\ 0, \text{H2A\_MAC\_REQUEST}),$
$\text{N\_A2CA}(\mathbf{n2w}\ 0, \text{A2H\_MAC\_GRANT}(\mathbf{n2w}\ 50), 84 * \text{NSEC})]$

V. Experimental Validation, Trace Checking

In order to check traces for conformance, we transform our specification to a checkable version, which we prove equivalent to the original. We then use the automatic code extraction facility of HOL to generate an ML program that checks that a trace conforms to our original specification. We use this in a distributed setting to check many traces in parallel.

Transforming the specification, and proving correctness The specification is not *directly* usable to test conformance of implementations for two reasons. First, the specification involves infinite sets, functions and other mathematical structures that may not be executable. For example, the specification may talk about two infinite sets being equal, whereas in general there is no way to check equivalence of infinite sets programmatically. Second, the specification talks about infinite traces, whereas observed implementation traces are finite.

The first problem can be solved by replacing logical expressions with equivalent expressions that are programmatic. For example, we can prove that all sets mentioned by the specification are in fact finite when considered in the context of a finite trace, and then replace expressions that talk about sets with equivalent expressions that talk about finite sets, which can be handled programmatically.

The second problem is more serious. First we define what it means for a finite trace to satisfy the specification, then we manually state an executable specification for finite traces and prove that it is equivalent to the original specification.

We say that a finite trace t_{fin} satisfies the specification iff the finite trace t_{fin} extended with an infinite sequence of null (τ) events satisfies the specification. There are reasonable alternatives, but this definition seemed the simplest. However, if an implementation trace ends prematurely (say, a host was about to reply to a ping), it may trivially fail to satisfy a property in the specification (say, that pings are followed by pongs). We therefore informally ensure that the traces we check, as far as possible, do not terminate until the network quiesces. In the case of non-quiescent properties, such as the requirement that every ping is followed by another, we relax the specification somewhat. For example, we require that every ping is followed by another *as long as time keeps increasing*.

We then manually write a transformed version of the specification which is executably checkable. We prove in HOL that the executable version is equivalent to the original, given our definition of what it means for a finite trace to satisfy the specification. Since *spec* is a essentially a conjunction of many properties P_i, it suffices to derive, for each property P, an equivalent executable property P'.

For example, suppose we wish to check the property that every ping is followed by a pong. In an infinite trace, a ping could occur anywhere, so to check this property would require checking an infinite number of points in the trace. Obviously, when dealing with a finite trace, only a finite number of points have to be checked. The correctness of this transformation rests on the simple observation that beyond a certain point, the finite trace extended with τs contains no pings.

Another example that arose very often relates to the way functions are defined. For example, the time at a point in the trace takes into account what has happened up to, but not including, the event at that point. Thus, to check whether some time corresponds to a point in a finite trace, one must check all points in the finite trace, and one point beyond the end of the finite trace, to take account of the possibility that the last event was a time passage event. When we manually write our checkable version of a property, this introduces numerous possibilities for "off by one" errors.

These examples are simple, but in general the correctness of the various transformations we make are far from straightforward, and the proofs are often significant. For example, some of the properties place constraints on several intervals in a trace. To prove the executable version equivalent involves several nested case splits on whether the intervals intersect the τ part of the extended finite trace. Such case splits can easily become unmanageable without mechanical support. The combination of several of these transformations produces even more complexity.

We made further transformations to make the checker more efficient, altering the logical structure of properties so that tests would pass or fail more quickly. More advanced transformations aim at precomputing various quantities to avoid unnecessary recalculation by the checker. These performance optimisations had a great impact. For example, early versions of the checker took over 5 minutes to check some properties of relatively short traces. This time was reduced to less than 30 seconds using fairly straightforward optimisations. Standard complexity analysis suggests that the checker runs in time which is polynomial in the length of the trace, typically of order $O(n^4)$ or less. We believe that this time could be made almost linear with further optimisations. However, the additional proof burden becomes significantly greater as the optimisations become more involved, and we can already check traces of reasonable length using the current checker.

We place great emphasis on the benefit of a formal proof of correctness for the checker. The specification contains many complicated properties, and the associated checker is even more complicated and convoluted because of the transformations above which are made for reasons, such as efficiency, which oppose clarity. Sometimes we ran the checker without formally proving it correct, and almost always it contained errors which could cause a trace to pass even though it did not satisfy the specification, or to fail even though it did satisfy the specification. For our final verified checker, these possibilities could be discounted because we formally proved that the checker did not have errors. This is born out by experience: of the many thousands of traces we checked, we never found an error in the verified checker code. This represents an advance

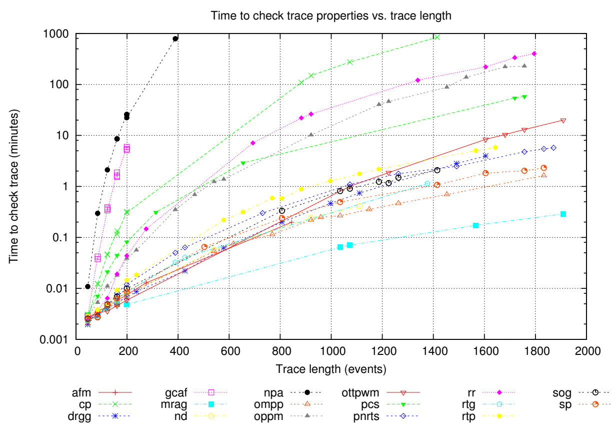


Fig. 3. Checker performance

over similar tools that were developed on our earlier TCP work [5], [6], where, if a trace did not pass the checker, it may have been due to an incompleteness or bug in the checker.

CODE EXTRACTION We have described how we transformed the specification into an algorithmically checkable version. We could execute this algorithm within HOL, using rewriting. For performance reasons, however, we use the automatic code extraction facility of HOL to produce an ML program which checks the executable specification on finite traces. This program is then compiled to produce a relatively efficient checker. If there is any doubt about the correctness of the extraction step, we can recheck the property within HOL, but much more slowly.

DISTRIBUTED CHECKER ARCHITECTURE, FORMAT OF RESULTS The checker validates 18 properties of a trace. Checking a trace is computationally intensive but naturally parallel: each property is checked independently with a separate invocation of the checker process. A distribution architecture shares the checking load amongst a set of Intel® Xeon™ 3.0GHz workstations. A central computer runs the distribution server. Clients connect to this to obtain a trace to check. The client then runs the checker over the trace file with the property specified by the server and sends back the checker output.

The server machine includes a web interface to control the traces being checked and to view the results of checker runs. A table presents the name of a trace file and 18 columns next to it corresponding to the properties being checked. These boxes are either green or red to show whether the property passed the checking process. When the box is red an annotated trace may be viewed where erroneous events are highlighted to help diagnose the problem with the trace.

The client also returns the time taken for each check run. This information is graphed for checker runs of a simple scenario and presented in Fig. 3. Each line on the graph corresponds to a checker property. The traces are generated from NS-2 simulations of networks of different sizes being configured with MAC addresses and performing start-of-day operations.

VI. TEST RESULTS

The trace checker was used to check many traces produced by the two different implementations of the protocol. To date our checking work has focused primarily on the NS-2 model. For this we generated a total of 574 traces, dealing with start-of-day operations, communication between hosts, removal and addition of hosts to and from the network and random loss of messages on the control channel. The specification describes 18 different host and arbiter properties. By the end of the checking process all 18 properties were shown to hold for all 574 generated traces.

For the VHDL implementation, at the time of writing we have generated and checked only a few traces, demonstrating that the process is feasible, but not yet aiming for broad coverage.

During development of the specification and implementations, a failing trace could mean either that the trace described a behaviour we did not wish to permit, due to some error in the implementation that produced the trace, or that the specification was incorrect, in that it did not assert what the protocol designer had in mind (or both). We discovered errors of both sorts. To give an intuition for how discriminating the test process can be, we give some examples below.

ERRORS IN THE IMPLEMENTATION TRACES Failures resulting from traces describing an illegal behaviour arose for two reasons: either because the implementation itself was flawed (which is the kind of error we would most like the process to detect); or the implementation might behave correctly but the instrumentation and transcription of its behaviour could be erroneous.

A number of errors of the first kind were discovered in the NS-2 implementation.

- The specification forbids sending a grant message to a host which has not yet successfully responded to a ping message with a pong, as, without a good RTT estimate, it is not possible to know how much to offset a Data Grant message for a particular switch fabric slot. Early versions of the arbiter employed an imprecise approach to scheduling grants, which was found to violate this property.
- A subtle timing error in the host logic was found which meant that hosts reissued requests slightly earlier than was legal according to the specification.
- The arbiter was found to to send pings incorrectly during startup.

Additionally, a small number of errors of the second type were revealed, mostly early on in the process, to do with incorrect recording of MAC and port identifiers and events being logged in the wrong order, leading to impossible behaviours.

ERRORS IN THE SPECIFICATION The testing process also revealed errors in the specification itself. For example, the property $\mathrm{only\_talk\_to\_ports\_with\_macs}$ restricts the arbiter to only send messages to ports which have been assigned MAC identifiers. To allow MACs to be assigned in the first place, the specification makes an exception to this rule, and allows A2H_MAC_GRANT messages to be sent to ports without an assigned MAC. However, after trace generating and testing, it became clear that A2H_PING messages must also be allowed, so that start-of-day host discovery could take place.

As a further example, the protocol requires that when a grant is issued by the arbiter, the switch should maintain the corresponding light path for the duration of a particular interval calculated from the RTT. As initially written, this interval could sometimes begin *before* the grant was issued (due to the addition of a safety margin), even though there is no way the data could arrive at the switch that early in the presence of any amount of RTT drift. The arbiter implementation quite reasonably did not set the light path this early, which was reported as a discrepancy by the checking process; accordingly, the specification was changed.

DEALING WITH ERRORS Initially it was the case that most failures occured in groups — all or almost all of the traces would exhibit a failure of the same property due to some fundamental error. After such gross errors had been eliminated, the remaining failures usually occured singly, being the result of a highly specific combination of circumstances.

If an implementation error was detected, the implementation was modified, and a new set of traces were generated and checked. If a specification error was detected, the property was rewritten, a new executable version was formally proved correct, and a new checker program was automatically extracted and rerun on the traces. This produced a cyclic workflow, where traces were checked, errors detected, specification/implementation corrected and trace checking rerun. This cycle was conducted relatively quickly. The interface to the trace checker indicated which property was failing and the points in the trace that it failed, and it was then a simple matter to decide whether the specification or the implementation was broken. Changing the specification, reproving an executable property correct, and extracting the checker could be carried out in an hour or two. Similarly, changes to the implementations could be carried out relatively quickly.

VII. Related Work

Related work on optically-switched networking has already been discussed in §II. On the formal side, there is a huge literature on many different techniques, including process calculi, IO automata, term rewriting, model-checking, etc., and we make no attempt to survey it here. To the best of our knowledge no other work uses a combination of proof and automated code extraction to generate oracles for protocol testing.

Only a rather small fraction of other formal work relates to non-idealised (real-world) protocols. We discuss formal work related to TCP, for example, in [5]. A different small fraction makes use of expressive logics and theorem proving techniques such as HOL. This includes work on security protocols, e.g. that by Paulson [15], but little on lower-level network issues, and little also on formalisation at design time. Notable exceptions include the work of Bargavan et al., using HOL and SPIN to verify properties of the RIP standard [3], and of Goodloe et al., using the Maude tool for formal simulation during the design of a protocol for setting up IPsec associations [8]. Hickey et al. proved properties of the Ensemble distributed communication layer using IO automata in NuPRL [9].

VIII. Discussion

In this section we reflect on this experiment in lightweight rigorous protocol design and mention some possible future work.

OUR EXPERIENCE We found, in designing the protocol, that the formal component of our design process lent it several advantages over a more conventional approach.

For one thing, the necessity of specifying the protocol in HOL notation meant that ambiguities and inconsistencies in design choices were revealed almost as soon as they arose. Examples of unanswered questions which were brought to our attention quickly due to formalisation included:

- how hosts were expected to discover one another's MAC addresses;
- whether it was legal for the arbiter to reuse MAC addresses which had been invalidated;
- how the arbiter should respond to a request to send data to a non-existent address (one version of the protocol mandated that the arbiter ignore the request and the host then continue to reissue it *ad infinitum*);
- how the arbiter was to correctly deduce which of its ping messages a particular pong was in response to, in order to obtain RTT estimates; and
- whether and how the arbiter or hosts could usefully respond to observing 'light' on a link (they respond to noticing 'darkness', i.e. losing the link, but noticing the reappearance of the light is not taken to be a sufficient indication that the other end of the link is functioning correctly).

Formalising the specification also forced us to consider whether particular intervals and timeouts in the sample implementation should be considered to be concrete requirements or merely one of a range of possibilities, and in the latter case, what the bounds of that range were.

Since changes made to the specification during the testing process had to be immediately inserted into the HOL specification, it was usually very obvious when some proposed 'fix' would have conflicted with or invalidated an existing requirement, and the consequences of rephrasing a particular rule or definition could be much more easily investigated in terms of their effect on the protocol as a whole. Additionally, when discussing what we believed to be invariants of the system, it was much easier to look at the specification than the code to determine whether such a property was actually guaranteed to hold.

TEST COVERAGE Any testing process is clearly only useful if the generated traces provide good coverage of the protocol behaviour. We believe that our NS-2 trace set gives reasonably good coverage, and the fact that testing has found subtle errors supports this, but they do not explore *all* interesting aspects of the protocol. Improving coverage further may require checking longer traces, which (as one can see from Fig. 3) may require improving the performance of the checker. That could be done in several ways: most simply by targetting a high-performance ML implementation such as MLton [1], and most interestingly by more sophisticated proof about a better-structured algorithmic specification.

COMPARISON WITH TRADITIONAL VERIFICATION Our approach is in contrast to much traditional work on formal verification, which focuses on proving correctness properties. The standard safety property for a protocol such as this would be that, between any two devices, the network behaves like a buffer. Ideally this property would be proven within HOL. A verification of this sort would greatly increase confidence in the protocol. However, such proofs are extremely time consuming, and require substantial skill on the part of the verifier. We believe that our approach, while it does not guarantee this kind of correctness property, provides substantial benefits and is relatively lightweight. It stresses not the relation between the correctness statement and the model, but between the model and the real world; we have demonstrated that it is practically feasible to apply it during design, while the protocol is changing. An investment in full verification may be appropriate at a slightly later stage, when some confidence in the correctness and implementability of the protocol has already been established.

On the other hand, simply *stating* a formal end-to-end correctness property may be worthwhile early on, as a precise characterisation of what a protocol is intended to achieve. We have outlined such properties for the SWIFT MAC protocol, but do not detail them here.

The approach we describe here is rather different from traditional model checking. In particular, we have been able to use the full expressiveness of higher-order logic to write the specification as clearly as possible, and have been able to validate the correspondence between specification and implementation with only mild changes (for instrumentation) to the implementations. It would be interesting to try model-checking approaches on this example to contrast the two in more detail.

SPECIFICATION COMPLETENESS A good specification should be in some sense *complete*: it should be tight enough that any implementation that matches it is satisfactory in practice, e.g. that it will interoperate with any other conformant implementation. Our testing process does not establish this, whereas proof of an end-to-end correctness property would. Pragmatically, it would be interesting to have unrelated teams build implementations based on the spec. This would establish useful confidence in its completeness, and also test whether our choice of formalism and idiom is sufficiently clear.

CONCLUSION We have demonstrated that a lightweight style of rigorous specification, with automated conformance testing, is feasible for new real-world protocol designs.

We did so using global trace-based idioms in HOL; building a conformance checker by a combination of HOL proof and code extraction (thereby establishing high confidence in the correctness of the checker); writing the specification with testing in mind; designing the specification and implementations hand-in-hand (so checking can find errors in both); and involving both semantics and systems researchers, so discussion during the process can find omissions and misconceptions early. This was for a protocol of moderate complexity, but there are no obvious difficulties with scaling it up to larger protocols. Indeed, our expectation is that the benefits over a purely informal approach would be disproportionately greater for more complex protocols.

ACKNOWLEDGEMENTS We acknowledge support from a Royal Society University Research Fellowship (Sewell), EPSRC grant EP/C510712, and APPSEM 2.

REFERENCES

[1] MLton. `http://mlton.org/`.

[2] ns-2. `http://www.isi.edu/nsnam/ns/`.

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

[4] A. Biltcliffe, M. Dales, S. Jansen, T. Ridge, and P. Sewell. SWIFT MAC protocol: HOL specification. `www.cl.cam.ac.uk/users/pes20/optical/spec.pdf`.

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

[6] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proceedings of POPL 2006 (Charleston)*, Jan. 2006.

[7] M. Dales, M. Glick, and D. McAuley. Considerations for Control Planes in High-Capacity, Low-Latency, Optically-Switched Interconnects. In *High-Performance Networking: The Terabit Challenge*, 2006.

[8] A. Goodloe, C. A. Gunter, and M.-O. Stehr. Formal prototyping in early stages of protocol design. In *Workshop on Issues in the Theory of Security*, 2005.

[9] J. Hickey, N. A. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In *Proc. TACAS, LNCS 1579*, 1999.

[10] D. Huang, T. Sze, A. Landin, R. Lytel, and H. L. Davidson. Optical Interconnects: Out of the Box Forever? *IEEE Journal of Selected Topics in Quantum Electronics*, 9(2):614–623, March/April 2003.

[11] T. Lin, K. A. Williams, P. V. Penty, I. H. White, M. Glick, and D. McAuley. Self-Configuring Intelligent Control for Short Reach 100GB/s Optical Packet Routing. In *Optical Fiber Communications Conference*, 2005.

[12] D. McAuley. Optical Local Area Network. In *Computer Systems: Theory, Technology and Applications*, pages 159–166, December 2003.

[13] E. Mohammed, A. Alduino, T. Thomas, H. Braunisch, D. Lu, J. Heck, A. Liu, I. Young, B. Barnett, G. Vandentop, and R. Mooney. Optical interconnect system integration for ultra-short reach networks. *Intel Technology Journal*, 8:115–128, 2004.

[14] M. Norrish and K. Slind. *HOL-4 Manuals*, 1998-2006. Available at `http://hol.sourceforge.net/`.

[15] L. C. Paulson. Proving security protocols correct. In *Proc. 14th LICS*, pages 370–381. IEEE, 1999.

[16] G. F. Roberts, K. A. Williams, R. V. Penty, I. H. White, M. Glick, D. McAuley, D. J. Kang, and M. Blamire. Monolithic 2x2 amplifying add/drop switch for optical local area networking. In *29th European Conference on Optical Communication*, September 2003.

[17] S. Verma, H. Chaskar, and R. Ravikanth. Optical Burst Switching: A Viable Solution for Terabit IP Backbone. *IEEE Network*, pages 48–53, November/December 2000.