Stenning's Protocol Implemented in UDP and Verified in Isabelle

Michael Compton

Computer Laboratory
University of Cambridge,
JJ Thomson Avenue,
Cambridge CB3 0FD, UK,
Email: `Michael.Compton@cl.cam.ac.uk`

Abstract

This paper is about the mechanical verification of UDP based network programs. It uses the UDP portion of a formal model of the Internet protocols TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). The model includes asynchronous message passing, message loss and host failure. The model is based around the sockets library, the primary API used for writing UDP and TCP based applications. This paper demonstrates that formal, machine-checked, proof is possible in the UDP model by presenting the proof of a safety property for an implementation of Stenning's Protocol. The protocol is implemented in a fragment of the OCaml language, using the sockets library for UDP network communication. The entire development including the safety proof is carried out in the proof assistant Isabelle; this assures soundness. Thus this paper demonstrates that it is possible to machine verify very concrete representations of distributed programs in a detailed semantics that accurately reflects the programs execution environment. Previously only abstract representations of this protocol have been machine verified. The proof, based on an implementation, provides a contrast to other verifications.

Keywords: theorem proving, distributed systems, formal verification.

1 Introduction

It is well known that it is difficult to design distributed algorithms correctly. There are many subtle ways in which they can fail. Thus computer scientists have sought out formalisms to describe and verify distributed algorithms. However, proofs of properties of distributed algorithms can be difficult and tedious. A number of published algorithms, even with proofs, have later been found to be incorrect. As a result, machine tools, model checkers and proof assistants, have often been used to demonstrate the correctness, or otherwise, of distributed algorithms (Chkliaev, Hooman & Vink 2003, Bhargavan, Obradovic & Gunter 2002, Simons & Stoelinga 2001, Stoelinga & Vaandrager 1999, Chkliaev et al. 2003)

This paper takes a complementary view – *that it is also difficult to implement distributed algorithms correctly.*

Any algorithm can be hard to implement correctly, but distributed algorithms also involve the subtle interaction of a number of components through a network, over which there is little or no control, making the implementation yet more complex. Even having a correct specification, or protocol, does not ensure that the implemented code will be correct. This can be for a number of reasons including that an implementation often bears little resemblance to an abstract protocol description. Further, implementations rely on concrete features of programing languages and operating systems, rather than abstract operations like *send message m from A to B*. These implementation details have to be taken into account in when considering the correctness of programs.

This paper discusses the verification of distributed systems at the level of programs rather than abstract protocol descriptions. I briefly review the model of Internet (UDP) communication presented by Wansbrough *et al.* (Wansbrough, Norrish, Sewell & Serjantov 2002, Serjantov, Sewell & Wansbrough 2001*a*). I then integrate a programing language semantics with the model. This semantics is a fragment of a real language so actual programs can be expressed. In this language I implement Stenning's Protocol and verify a safety property with reference to UDP and the sockets interface. The main contribution of this paper is to demonstrate the verification of distributed program code in a model that accurately represents the operating system API and network environment in which the program would execute. [1]

UDP and the sockets interface. Distributed programs require some sort of underlying communication support. This support gets a message from one program, out onto the communication mechanism, possibly through various intermediate networks, and finally delivers the message to its destination. On the Internet two of the major options are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). These and a number of other protocols form what is ofter referred to as the Internet protocol suite. This paper considers the protocol UDP (Postel 1981, Postel 1980).

UDP provides an unreliable datagram service. This means that message transmission is on the level of individual packets or datagrams (not streams as in TCP), and that packets may be lost or reordered by the network. Further no connection information is maintained, nor is any state information retained about a message once it has been sent. For this paper all that need be known about UDP is that messages sent using UDP may be lost, reordered and duplicated; and that the sender of the message does not know what has happened to it.

[1] Note that no model can completely capture a real program's execution: for that you would need to model and verify everything from CPU to operating system to compiler to program. However, by closing the 'semantic gap' between verification and the executable system, we get a better assurance that the running program is correct.

A number of important applications are implemented using UDP, such as DNS (Domain Name System), DHCP (Dynamic Host Configuration Protocol) and RIP (Routing Information Protocol).

UDP based distributed programs are written using the sockets interface. Originally from BSD Unix, sockets are now part of the programing API on Windows and Unix operating systems. The sockets interface allows a programmer to open sockets of various forms and to send and receive messages between programs distributed across a network. The standard guide to sockets and Unix network programing is provided by Stevens (1998).

In this paper I consider the semantics of UDP and the sockets interface developed by Wansbrough *et al.* (Wansbrough et al. 2002). This semantics includes the various types of failure present in UDP networks: message loss, message duplication, crash of hosts, and disconnections of hosts from the network. It also involves time and therefore programs that rely on failure, for example those that use timeouts, can be reasoned about.

The definitions considered in this paper are very large: the sockets library definition alone is an inductive definition with over 70 rules and involves a number of complex functions. In order to manage such a definition, and to carry out proofs, machine assistance is required. The UDP model was defined as a theory in the proof assistant HOL (Gordon & Melham 1993). HOL assures that the definition is type correct and allows various meta properties to be proved. The model has also been tested against Linux and Windows implementations. This paper is concerned with a version of the model imported into the Isabelle proof assistant (Nipkow, Paulson & Wenzel 2002). This is not due to any deficiency with HOL, but simply because I am more familiar with conducting proofs in Isabelle.

Isabelle. All the definitions and proofs required for the results in this paper were carried out inside Isabelle (Nipkow et al. 2002). Isabelle is an interactive proof tool in the LCF tradition (Gordon, Milner & Wadsworth 1979). Such proof tools take a definitional approach, building all new theorems from previously proved results. The definitional approach is designed to assure soundness.

Isabelle supports a number of logical formalisms. Here Isabelle's instantiation of Higher Order Logic is used, called Isabelle/HOL (Nipkow, Paulson & Wenzel 2003, Nipkow et al. 2002). This system is quite different to HOL, but the two support a similar logic.

I try as much as possible to avoid a reliance on Isabelle syntax. All theorems and definitions are explained when presented, using as clear syntax as possible. The reader need not be familiar with Isabelle and need only know that the work is supported by a mechanical tool.

MiniCaml. The UDP semantics is language independent. It includes a notion of threads but does not restrict the language that the threads are implemented in. This paper discusses the development of a semantics for a fragment of OCaml, called MiniCaml.

The MiniCaml and UDP semantics are integrated so that networks of distributed MiniCaml programs can be defined in the formal model. As MiniCaml is a subset of OCaml I can express programs that can be compiled using the OCaml compiler. These compiled programs can then be tested in real networks. The program text can also be imported into the formal model and reasoned about. In particular I consider the implementation and verification of Stenning's protocol.

Together the UDP and MiniCaml semantics define a large labelled transition system. During a computation the network will emit a label for every action performed; a list of such labels forms a trace. This paper is concerned with verifying that every trace of a network satisfies some safety property. The systems considered are commonly called *reactive systems*: they interact with their environment and react to external input. Hence this paper is also about the verification of reactive systems, using a trace based model of computation.

Paper outline. The paper begins by outlining the UDP model (Section 2), including the semantics of the sockets library (Section 2.2). It then discusses the MiniCaml semantics (Section 3). Stenning's protocol is described (Sections 4 and 4.1) and a MiniCaml implementation is presented (Section 4.2). The main result of this paper is the mechanical verification of a safety property of the Stenning protocol implementation (Section 5). Finally, a concluding section (Section 6) evaluates this work and discusses possible improvements and extensions.

2 UDP and Network Model

In this paper, a network consists of three main components: messages in transit, hosts, and threads. Specifically a network is generated by the following grammar.

$N =$	ϵ	:	The empty network.
	$N \mid N$	:	Parallel composition of two networks.
	$\mathit{Msg}\ m$	:	A message in transit.
	$\mathit{HC}(n, \mathit{comp})$	:	A component of host n.

Messages have a simple internal structure, closely following the structure given the RFC (Postel 1980). The important elements are its source, destination and a block of message data.

Hosts are the most complicated network component. A host is composed from a number of network components. Here host components will either be $\mathit{HC}(n,\ \mathit{Host}(b, h))$, for the actual state of host n (open ports, messages queued to be sent and received etc.), or $\mathit{HC}(n,\ \mathit{Thread}(\ldots))$, for a thread running on host n. A complete host will be the composition of a number of interacting network components. Only networks passing certain sanity properties can be reasoned about effectively; for example, networks containing two copies of a single host's state are clearly not sensible. As in an actual network, a host will be identifiable to other hosts by some IP addresses.

The dynamic semantics of a network is defined by three transition relations. The relations are summarised in Figure 1.

- The host semantics describes the state changes on an individual host. These changes occur due to incoming messages, the passage of time and calls to socket library functions. Socket library calls may create new sockets, queue messages to be sent and deliver messages to running threads. The relation is described in Section 2.2.

- An execution on a host is called a thread, many threads may execute on a single host. In this paper threads represent the execution of MiniCaml programs. The semantics of MiniCaml (the dynamic portion of which constitutes the thread semantics) is discussed in Section 3.

Host Semantics

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

Thread Semantics

$$t \overset{l}{\twoheadrightarrow} t'$$

Network Semantics

$$\frac{h \xrightarrow{\;\;l_h\;\;} h'}{HC(n, Host(b, h)) \xRightarrow{l} HC(n, Host(b, h'))}$$

$$\frac{t \overset{l_t}{\twoheadrightarrow} t'}{HC(n, Thread(\ldots, t)) \xRightarrow{l} HC(n, Thread(\ldots, t'))}$$

$$\frac{dpropmin \leq d \wedge d \leq dpropmax}{\epsilon \xRightarrow{(Lb\ (n, In, L\_sendmsg\ m))} (Msg(Timed(m, d)))}$$

$$\text{Net\_Sync}\,\frac{N_1 \xRightarrow{l_1} N_1' \quad N_2 \xRightarrow{l_2} N_2' \quad sync\ l_1\ l_2\ =\ Some\ l}{N_1 \mid N_2 \xRightarrow{l} N_1' \mid N_2'}$$

Figure 1: Summary of the three labelled transition systems (semantics) that make up the entire UDP model. The host semantics (→) describes the state changes on a host, it defines how calls to the sockets library affect hosts. The thread semantics (↠) corresponds to the dynamic part of the MiniCaml semantics. The network semantics (⇒) combines the host and thread semantics, describing the interaction (synchronisation) of hosts and threads. It also describes how messages traverse the network. The network semantics is non-deterministic allowing for a number of possible execution options.

- These two disjoint semantics are bound together by a transition relation for the network semantics. This relation describes message propagation, failures (host crashes, message loss etc.), the passage of time, and the interaction of hosts and threads. There are over 50 rules that describe the network transition relation. Section 2.1 discusses the network semantics and how it links the host and thread semantics.

As Figure 1 indicates, all three relations are small step operational semantics defining labelled transition systems. Each is defined inside Isabelle, using its mechanisms for inductively defined sets. This mechanism makes sure the definition satisfies certain monotonicity properties and generates an induction principle.

2.1 Network Semantics

The rules for the network labelled transition system are of two types. One defines the actions of individual network components, the other describes synchronisation between networks of (possibly) many components built with parallel composition. The first type of rule is shown in Figure 1 by the two rules showing that host and thread actions can be lifted into the network and by rules for messages. The second type of rule is shown by the rule Net_Sync. Let us consider how these rules work together to describe network execution.

Figure 1 gives a rule that shows how a message may become a network component; it is labelled by *L_sendmsg m*. In this rule m is the actual UDP message, which is wrapped in time information so that the model can assure messages are delivered within certain time constraints. As m is unconstrained this rule says that if the propagation time obeys some constraints then any message may appear on the network.

It may seem strange that arbitrary messages can 'invent themselves' and enter the network. This problem is solved by the rules lifting host actions into the network and the Net_Sync rule. There are many instances of rules similar to the rule for hosts shown in Figure 1. They describe the different ways in which host actions can affect the network. For example, the following is the rule for when a host sends a message.

$$\frac{h \xrightarrow{Lh\_sendmsg\ m} h'}{HC(n, Host(True, h)) \xRightarrow{(Lb\ (n, Out, (L\_sendmsg\ m)))} HC(n, Host(True, h'))}$$

This rule says that if a host can send a message (the transition above the line is part of the host semantics) then the network component consisting of only that host can send the same message. Here the host component $HC(n, Host(True, h))$ is simply the state information h, for host n: remember that the state information includes messages queued to be sent. We would expect that message m is on an outgoing queue of h; perhaps a thread had previously asked for it be sent using a socket library call.

The important thing to notice about these two *L_sendmsg* rules is that they define the two halves of one action. Note that the labels are identical except for *In* and *Out*. These two rules need to be synchronised, outputting a *Tau* label. This sort of synchronisation algebra is familiar in many areas of concurrency theory, such as versions of the π-calculus (Milner 1999). The Net_Sync rule defines how these two rules can synchronise and output the label $Lb(n, Tau, (L\_sendmsg\ m))$. In a complete network, neither rule would be allowed to fire independently. Only the synchronisation of the two could happen, thus constraining the messages actually sent For example, for some arbitrary network N the following transition could occur.

$$\begin{gathered} HC(n, Host(True, h)) \mid \epsilon \mid N \\ \xRightarrow{(Lb\ (n, Tau, (L\_sendmsg\ m)))} \\ HC(n, Host(True, h')) \mid Msg(Timed(m, d)) \mid N \end{gathered}$$

In this way the actions of individual components of the network are lifted to actions of the whole network and forced to be consistent. Note that only the network labels are emitted; the actions of the participants, here the host label *Lh_sendmsg m*, will be forgotten. Also it is not known if the message will be lost or delayed, only that a host asked for it to be sent. Hosts and threads synchronise in a similar way for socket library calls.

In this model, networks contain two types of synchronisation. Synchronous communication occurs *socket_1* **<u>succeed</u>**

$$h \text{ with } \mathit{ts} := \mathit{ts} \oplus (\mathit{tid} \mapsto \textsc{Run}_d)$$

$$\xrightarrow{\mathit{tid}\cdot(\mathsf{socket}())}$$

$$\begin{aligned}&h \text{ with } \langle\!\![\mathit{ts} := \mathit{ts} \oplus (\mathit{tid} \mapsto \textsc{Ret}(\mathrm{OK}\mathit{fd})_{\mathit{dsch}});\\ &\quad s := (\textsc{Sock}(\mathit{fd}, *, *, *, *, *, \textsc{Flags}(\mathbf{F}, \mathbf{F}), [\,]) :: h.s)]\!\!\rangle\end{aligned}$$

$\mathit{fd} \notin \text{sockfds}\, h.s$

Figure 2: This rule, from the host semantics, describes the state change on a host when a new socket is created. The rule says that if there is a running thread on the host then the host may create a new socket. The rule emits a label indicating that it performed the sockets library call `socket()`. The new socket is added to the hosts list of sockets, the side condition says that the socket must have a file descriptor that is not currently in use. Of course in a network this rule would need to be synchronised with a thread that also wants to make a call to `socket()`.

recvfrom_1 **<u>succeed</u>**

$$\begin{aligned}&\mathit{FC}(\mathit{ifds},\\ &\quad \mathit{tid}, \textsc{Run}_d,\\ &\quad s \text{ with } \langle\!\![\mathit{ps}_1 := \uparrow p_1; \mathit{es} := *;\\ &\quad \mathit{mq} := (\mathrm{IP}(i_3, i_4, \mathrm{UDP}(\mathit{ps}_3, \mathit{ps}_4, \mathit{data})), \mathit{ifid}) :: \mathit{mq}]\!\!\rangle)\end{aligned}$$

$$\xrightarrow{\mathit{tid}\cdot\mathsf{recvfrom}(s.\mathit{fd},\mathit{nb})}$$

$$\begin{aligned}&\mathit{FC}(\mathit{ifds}, \mathit{tid}, \textsc{Ret}(\mathrm{OK}(i_3, \mathit{ps}_3, \mathit{data}))_{\mathit{dsch}},\\ &\quad s \text{ with } \langle\!\![\mathit{ps}_1 := \uparrow p_1; \mathit{es} := *; \mathit{mq} := \mathit{mq}]\!\!\rangle)\end{aligned}$$

F_context FC

Figure 3: This rule, from the host semantics, describes one case for the socket library call `recvfrom()`. $\mathit{FC}(i, \mathit{tid}, \mathit{ts}, s)$ represents a host with the constrained parameters i, the thread identifier *tid*, the current state of the thread *ts*, and the socket s. Here a host with a UDP message in the message queue *mq* of socket s may pass that message on to a thread. Again in a network this action would need to be synchronised with the corresponding thread.

as above. There is also asynchronous communication. Threads on different hosts communicate asynchronously through the UDP messages that traverse across the network.

Network execution is nondeterministic. In a network, any of the available actions could be nondeterministically selected and performed, ignoring other possible actions. For example, in the network above if N could perform some reduction step then this step could have be chosen instead of the step illustrated. This gives the notion of interleaving and nondeterminism required for concurrency semantics.

2.2 Host Semantics

The labelled transition system for hosts is somewhat larger than the network semantics. This is the result of the complexity of state information and the large number of socket calls to be dealt with. A version of the entire host semantics has been presented in a technical report (Serjantov, Sewell & Wansbrough 2001*b*).

The semantics of hosts is described by over 70 rules; this definition alone, not counting any auxiliary functions, is over 1000 lines of theory text. These rules set out precisely the behaviour of the sockets interface. The rules were developed from inspection of Linux kernel sources and existing documentation such as Stevens (1998). A large number of functions are also required in the definition.

Figures 2 and 3 give some idea of the semantics. These are two of the simplest rules, and thus easiest to explain. Each host rule contains 3 parts. Each rule has a label and a description of the success or otherwise of the socket call – any socket library call can succeed or fail in a surprisingly large number of ways: there are 10 rules describing the `sendto()` call. The central part of the rule describes the transition. Side conditions for the transition are listed at the bottom.

A host's state consists of the identifiers and some status information for all currently running threads (not the actual thread itself, this will be another host component in the network), a list of open sockets, and a queue of outgoing messages. Incoming messages are queued in the socket at which they arrive. State changes in a host are usually the result of a socket library call, but can also be the result of interaction with the network, as was the case in the transitions explained in the previous section.

Figure 2 presents the rule for creating a new socket, using a call to the function `socket()`. The transition of this rule states that any host containing a thread with status *RUN* may perform this transition. The result is to add a new socket $\mathit{SOCK}(\mathit{fd}, \ldots)$ to the host; the threads state is changed to indicate that it should be returned the file descriptor *fd* of the socket. The side condition indicates that the file descriptor should be new.

Again, this rule seems to say that a host may arbitrarily create sockets for its running threads. However, in a network a host would need to synchronise with a thread actually making a call to `socket()` in order to use this transition.

Figure 3 presents one of the successful rules for `recvfrom()`. A call to `recvfrom(fd,...)` asks to return the first message from the socket associated with `fd`.

3 MiniCaml Semantics

OCaml is a strongly typed functional programming language in the ML tradition (Xavier Leroy et al 2004). Here a fragment of OCaml, MiniCaml, is considered. MiniCaml programs can be compiled by the OCaml compiler and executed; UDP programs can be written, compiled and executed in a real network. A simple interface to the OCaml sockets library was written that makes the MiniCaml sockets library look exactly like the sockets library from the semantics above. The text of MiniCaml programs can be incorporated into the formal model via a simple syntax translation. Currently this translation is done by hand, but a program could perform this task.

This paper considers MiniCaml programs written using expressions as described in the following.

$$
\begin{array}{rl}
e = & (e,\ e) \\
 & (e\ ::\ el) \\
 & \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
 & \textbf{ref}\ e \\
 & !\ x \\
 & x\ :=\ e \\
 & \textbf{while}\ e\ \textbf{do}\ e \\
 & e;\ e \\
 & f\ e \\
 & \textbf{let}\ pat\ =\ e\ \textbf{in}\ e
\end{array}
$$

MiniCaml contains tuples $(e,\ e)$ and lists $(e\ ::\ el)$. **if** e **then** e **else** e and **while** e **do** e are the familiar choice and looping constructs. **ref** e, $!\ x$ and $x\ :=\ e$ allow references to mutable values to be created in the program. $e;\ e$ represents sequencing and $f\ e$ is the application of function f to expression e. Finally **let** $pat\ =\ e1$ **in** $e2$ evaluates expression $e1$, matches the result with pat, and then executes $e2$ with identifiers in $e2$ substituted for the values obtained from the evaluation of $e1$. This represents pattern matching familiar from many functional languages. Patterns can be empty, identifiers (which in the case of **let** will be bound in the following expression), constants, pairs, lists, and explicitly typed variants of any pattern. All these expression have the same meaning in MiniCaml programs as in OCaml programs.

The MiniCaml semantics needs to be a labelled transition system, where the labels interact in appropriate way with the host and network semantics outlined in the previous sections.

Most actions of programs should not be visible at the network level. For example, internal computations and decisions should not be observable in the network. We should be able to observe that the program is doing something but not know exactly what, except when it interacts with its environment. In this case environmental interaction is generally via socket calls.

Internal computations are modelled with rules that emit tau labels, while external rules will emit labels that when lifted into the network semantics (see Figure 1) will synchronise with a host. Figure 4 shows some of the rules from the MiniCaml semantics. Many of the rules are as expected for any language similar to this. They either perform some internal step, emitting a tau, or perform some unknown step and simply emit the corresponding label.

Socket calls emit the most interesting labels. The socket call at the bottom of Figure 4 shows the general case. Consider the function f to be any function from the sockets interface, say `socket()`. In this case the the label emitted by the reduction of this expression will be

$$Lt\_callHostOut\ (socket\ ()).$$

After this rule and an instance of the rule in Figure 2 have been lifted into network semantics, they could synchronise. In this case, a host component and a thread component would synchronise and emit the label $Lb(n, Tau, (L\_callHost(tid,\ socket())))$. This would indicate that the thread identified by tid and running on host n has made the $socket()$ call.

MiniCaml programs do not need to be part of a network; we can consider them in isolation. Expressions emitting tau labels will behave the same in this interpretation. However, expressions interacting with the network will have an interesting interpretation. Consider the following transition

$$
\begin{array}{c}
recvfrom(fd, false) \\
\xrightarrow{Lt\_callHostOut\ (recvfrom\ (fd, false))}\!\!\twoheadrightarrow \\
Ret_{recvfrom}
\end{array}
$$

$$
\textbf{if}\ \mathit{True}\ \textbf{then}\ e1\ \textbf{else}\ e2 \overset{\tau}{\twoheadrightarrow} e1
$$

$$
\textbf{if}\ \mathit{False}\ \textbf{then}\ e1\ \textbf{else}\ e2 \overset{\tau}{\twoheadrightarrow} e2
$$

$$
\frac{eb \overset{l}{\twoheadrightarrow} eb'}{\textbf{if}\ eb\ \textbf{then}\ e1\ \textbf{else}\ e2 \overset{l}{\twoheadrightarrow} \textbf{if}\ eb'\ \textbf{then}\ e1\ \textbf{else}\ e2}
$$

$$
\frac{e1 \overset{l}{\twoheadrightarrow} e1'}{e1\ e2 \overset{l}{\twoheadrightarrow} e1';\ e2} \qquad ();\ e2 \overset{\tau}{\twoheadrightarrow} e2
$$

$$
\frac{e1 \overset{l}{\twoheadrightarrow} e1'}{\textbf{let}\ p\ =\ e1\ \textbf{in}\ e2 \overset{l}{\twoheadrightarrow} \textbf{let}\ p\ =\ e1'\ \textbf{in}\ e2}
$$

$$
\frac{match(v, p)\ =\ \mathit{Some}\ sl}{\textbf{let}\ p\ =\ v\ \textbf{in}\ e \overset{\tau}{\twoheadrightarrow} (subs_list\ sl\ e)}
$$

$$
f\ v \xrightarrow{Lt\_callHostOut(f\ \ v)} Ret_f
$$

$$
Ret_f \xrightarrow{Lt\_RetHostIn\ \ v} v
$$

Figure 4: Some of the rules from the MiniCaml semantics. Many of the rules emit a tau label indicating an internal action. Socket calls will emit labels describing the call made. When a MiniCaml program executes in a network socket calls will be required to synchronise with a host also making the appropriate call.

According to the semantics in Figure 4, $Ret_{recvfrom}$ is ready to receive a message from the network. However, if we are considering this program in isolation there is no network. At this point nondeterminism is introduced: the program could choose any possible message to input and continue with. Some authors call this behaviour local guessing of network values (de Roever, de Boer, Hannemann, Hooman, Lakhnech, Poel & Zwiers 2001). At first this nondeterminism may seem to complicate matters, but it is in fact useful in verification. If we can show that a program has a certain property in isolation, i.e. for all possible choices of unknown values, then we know it will have that property in any network. Our verification method for distributed UDP system consists of two stages: first we show programs satisfy certain properties in isolation, then in a network show that the interaction of these properties produces the desired result for the network.

MiniCaml has a sensible type system, which is also encoded in Isabelle. Using the entire language (expressions, type system and LTS semantics) we can prove that a range of sanity properties hold of the language. Mostly these properties are ones that could be expected of any reasonable language. Only two are presented here. Type preservation tells us that as the program executes expressions do not change type.

$$
\text{Type Preservation}\ \frac{\begin{array}{c} e \xrightarrow{l} e' \\ \Gamma\ \vdash\ e\ :\ T \end{array}}{\Gamma\ \vdash\ e'\ :T}
$$

Also any well typed program is either a value, an error or some rule can be found which reduces the expression and emits a label.

$$
\text{Progress}\ \frac{\Gamma\ \vdash\ e : T}{is\_value\ e\ \lor\ is\_error\ e\ \lor\ \exists\ l\ e'.\ e \xrightarrow{l} e'}
$$

The subset of the language that emits tau labels is also deterministic. These properties simply tell us that the language is working properly. They are proved by induction over the appropriate definition, the structure of expressions, the typing relation, or the transition system. Mostly the proofs are not difficult but require care as as a number of the definitions are of mutually inductive sets.

4 Stenning's Protocol

As an example of verifying programs in the UDP model, let us consider the verification of an implementation of Stenning's Protocol. This protocol is designed for networks which (just like UDP) may lose, reorder and duplicate messages. The protocol assumes that received packets are not corrupted; our UDP model makes the same assumption, relying on lower level checksums to handle packet corruption issues.

First the protocol is described and an abstract description is presented. Then a MiniCaml implementation is given.

4.1 The Protocol

Stenning's protocol solves the problem of implementing a reliable FIFO channel between two processes connected by a network which may lose, reorder and duplicate messages. The protocol consists of two parties, a sender and a receiver, connected by some unreliable network. The sender wishes to send a message to the receiver with the assurance that the message will not be lost or reordered.

Consider two network nodes, N_S (the sender) and N_R (the receiver). Stenning's protocol involves N_S and N_R performing the following actions. N_S splits the message into a number of packets $p_1, \ldots, p_n$ (UDP imposes a restriction on the size of packets, so not all messages could be sent in one packet.) N_S then repeatedly sends packet p_1 to N_R and awaits an acknowledgement that the packet has arrived. N_R waits to receive a copy of p_1. When the first copy arrives, N_R accepts the packet and sends N_S an acknowledgement for p_1. If any packets arrive out of order, N_R repeats the current acknowledgement. Upon receiving an acknowledgement for packet p_1, N_S knows that N_R has received p_1 and is waiting for p_2. So N_S moves on to repeatedly sending packet p_2. The process is repeated for all packets.

Figure 5 shows one possible run in the operation of the protocol. It may be intuitively clear that the messages accepted by N_r are in order. Despite the unreliable channel connecting N_S and N_r they have implemented a reliable FIFO channel at the level of the packets $p_1, \ldots, p_n$. See Lynch (Lynch 1996) for further discussion of the protocol and a proof of its correctness. Section 5 will discuss verifying that the implementation presented next meets similar properties.

4.2 Implementation

Figure 6 presents the MiniCaml implementation of the sending side of the protocol, the receiving side is omitted. Rather than send a packet number and a message, the program only sends the packet numbers: there are no other sensible messages to send with the packet numbers. A complete program would be different: it would break up some larger message and send it in numbered packets to be reconstructed at the receiving end. However, the example considered here will be enough to demonstrate that we can reason about such programs. Section 6 further discusses this program and suggest improvements.

The MiniCaml implementation works as follows. The section labelled 1 simply creates sockets and prepares the program to send and receive data; this section is not needed in any protocol description but is of course essential in a real program. The code below 2 actually implements the protocol.

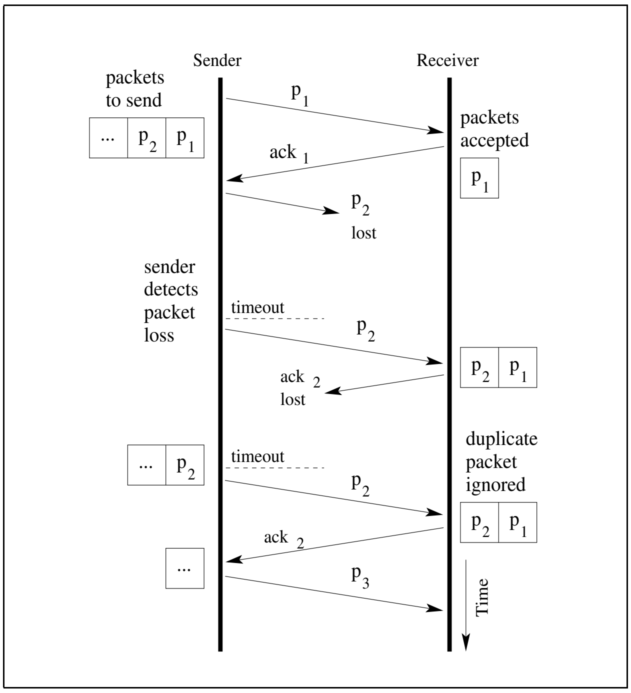

Figure 5: Shows a possible execution of Stenning's protocol. The Sender repeatedly sends each message until an acknowledgement is received and then moves on to the next message. Note that despite network error, messages will be accepted in order by the receiver.

The outer while loop of 2 is executed until all messages have been sent; `count` indicates the message currently being sent. At 3, the current message is initially sent. The inner while loop, labelled 4, corresponds to repeatedly sending a packet until it is acknowledged. The sockets library function `select()` is used to test if any messages have been received. Given a list of sockets and a timer, a call to `select()` will wait until either the timer expires (returning an empty list) or a message arrives on one of the sockets (when it will return a list of sockets on which messages are waiting). Intuitively, the while loop waits until a message arrives in the socket `fd_recv` or 1000000 microseconds have passed. If the timer expires, it is assumed that the message was lost and needs to be resent. This is an example of using a timer to indicate failure.

When the code labelled 5 is reached, there is definitely an message waiting in the socket `fd_recv`: the while loop would not have terminated otherwise. The message still has to be checked to ensure it is the acknowledgement for the current message. It may of course be some delayed or repeated acknowledgement, in which case it should be ignored. If it is the correct acknowledgement then `count` is increased and the program moves on to sending message `count + 1`.

The program for the receiving side is actually less complex. It simply waits for a message and acknowledges message receipt if the received message is in the correct sequence.

This program is a valid OCaml program and thus can be compiled and executed in a network. The program text can also be embedded as a MiniCaml program in the formal model described in Sections 2 and 3. We now turn to the question of mechanically verifying that a network consisting of the sending and receiving MiniCaml programs satisfies and appropriate safety property.

```
try

(* --- 1 --- *)
  let num_to_send = 20 in

(* set up to send messages *)
  let p_send = port_of_int 6666  in
  let i_send = ip_of_string "128.232.9.15"  in
  let fd_send = socket () in
  let _ = connect (fd_send,i_send,Lift p_send) in

(* set up to receive ACKs *)
  let p_recv = port_of_int 7777 in
  let i_recv = ip_of_string "128.232.9.37" in
  let fd_recv = socket () in
  let _ = bind (fd_recv, Lift i_recv, Lift p_recv) in

(* loop and send all messages *)

(* --- 2 --- *)
  let count = ref 0 in
  while !count < num_to_send do

    (* --- 3 --- *)
    sendto (fd_send,Star,string_of_int !count,false);

    (* --- 4 --- *)
    while (let (fds,_) = (* 4 *)
      select ([fd_recv],[],Lift 1000000) in fds = [])
    do
      sendto (fd_send,Star,string_of_int !count,false);
    done;

    (* --- 5 --- *)
    let (_,_,m) = recvfrom (fd_recv,false) in
    if int_of_string m = !count
    then
      count := !count + 1
    else
      (* Packet arrived out of order, ignore it. *)
  done

with _ ->
  print_string "Error!!!";
  print_newline ()
;;
```

Figure 6: The MiniCaml implementation of the sending side of the Stenning Protocol.

5 Formal Verification

This section outlines the formal proof of the implemented protocol. Proving that the protocol is correct means showing that all possible executions of a network containing the sending and receiving programs satisfy some sensible property. This section begins by discussing a suitable representation of computation, developing a notion of traces. It then discusses appropriate properties to prove and finally outlines the proof conducted in Isabelle.

Traces. Consider a network N that performs some number of steps $N \xrightarrow{l_1} N_1 \xrightarrow{l_2} \dots \xrightarrow{l_n} N_n$, according to the network transition relation. These steps define a computation performed by the network. The computation starts in state N, finishes in state N_n, and emits the labels $l_1,\ l_2,\ \dots,\ l_n$. This list of labels $[\ l_1,\ l_2,\ \dots,\ l_n\ ]$ is a trace. The trace is a record of the computation performed by the network. Some of the labels in a trace will not be interesting; tau actions for example will say nothing about what happened. Other labels will describe messages sent and received or host crashes, which are more interesting for verification.

A trace could have also been defined as the list of states a network goes through. In the case above the trace could have been $[\ N,\ N_1,\ \dots,\ N_n\ ]$. Both forms of a trace are acceptable. I felt that traces of labels would be easier to reason about, as the state of an entire network is large and complex. Traces of labels (sometimes called actions) are very simple, but contain enough information to reason about.

Traces are easy to define in the UDP network model. The following inductive definition yields the definition of network traces, and an induction principle.

$$(1)\ \frac{}{N \xRightarrow{[]} N}$$

$$(2)\ \frac{N \xRightarrow{tr} N';\ N' \xrightarrow{l} N''}{N \xRightarrow{tr@[l]} N''}$$

Definition (1) tells us that the empty list [] is a trace. Definition (2) says that if tr is a trace from N to N', and if l is the label of a transition from N' to N'', then $tr@[l]$ (where @ appends two traces) is a trace from N to N''. We can define a similar concept of traces for MiniCaml programs.

Due to the nondeterminism inherent in network execution, one network state could legally produce a number of traces. We will want to verify that all of these possible traces satisfy the safety property. This can be done using the induction principle generated by the definition above. Assume that we wish to prove that all traces of network N satisfy property P. The induction principle generated by the definition above has two cases. The base case asks us to prove that P holds of the empty list. The step case asks us to prove that if P holds of some trace tr then P also holds of all possible extensions l to tr.

The initial state. This proof will consider the traces of a simple network involving only the two programs from the Stenning protocol and two hosts. If *stenning_send* is the code in Figure 6 and *stenning_recv* is the receiving side of the protocol, then the initial state of sending and receiving hosts could be described as the following networks.

$N_S =$
$HC(id_{send}, Host(True, (simple\_host\ sender\_ip)))\ |$
$HC(id_{send}, Thread(Timed((\dots, stenning\_send), d)))$

$N_R =$
$HC(id_{recv}, Host(True, (simple\_host\ receiver\_ip)))\ |$
$HC(id_{recv}, Thread(Timed((\dots, stenning\_recv), d)))$

The details of each component are not important; *simple_host* creates a simple host state given an IP address for the host. N_S is simply a host with the *stenning_send* program ready to execute on the host, similarly for N_R and *stenning_recv*. Network composition is both associative and commutative, so networks may be composed in any order. The initial state of the entire network is then given by

$N_{initial} = N_S \mid N_R$

The property. One of the important safety properties for Stenning's protocol is that it implements a FIFO channel between sender and receiver. Here I show that messages sent from sender to receiver are sent and accepted in order. This seems seems to be the equivalent property for this version of Stenning's protocol.

Definition 1.
stenning_property tr $=$

$\forall\ i\ m.\ \mathit{accepted\ receiver\ m\ tr\ i}\ \longrightarrow$
$((\exists\ j\ <\ i.\ \mathit{sent\ sender\ m\ tr\ j})\ \wedge$
$(\forall\ n\ <\ m.\ \exists\ j\ <\ i.\ \mathit{accepted\ receiver\ n\ tr\ j}))$

Here *accepted receiver m tr i* says that the packet numbered m was accepted by the *receiver* at point i on trace tr, similarly for sent and received. Many messages will be received but only messages arriving in order will be accepted. This property states two things. Firstly, that if message m is accepted then all previous messages have already been accepted, assuring in order acceptance of messages. Secondly it states that that messages must come from the *sender*. To show that this property holds of all traces of $N_{initial}$ we need to prove Theorem 1.

The Proof. The proof is compositional. Rather than proving Theorem 1 in one large proof, it is better to prove properties of the individual network components, compose these components together and then use the properties of the components in proving the composed system. This compositional approach should require less work than trying to derive the final property in one large proof.

The proof technique has the following steps. Firstly, prove properties of the MiniCaml programs in isolation. This means prove that all traces of the program satisfy a property, say P_{ts} for the sending program. Compose the network N_S, giving the sending half of the final network. Derive, using P_{ts}, a property that holds for N_S, say P_S (a similar chain of reasoning is performed for the receiving half of the network). This requires reasoning at two levels in the semantics as P_{ts} is a property of MiniCaml traces and P_S is a property of network traces. Finally show, with the UDP messages sent between the two halves of the network, that the final property holds of all traces of $N_{initial}$.

All traces of the MiniCaml program for the sending side of the protocol satisfy the property in definition 2.

Definition 2.
thread_send_property tr =
$\forall\ i\ m.\ \ \mathit{sent}_{mc}\ m\ tr\ i\ \ \longrightarrow$
$(\forall\ n\ <\ m.\ (\exists\ jr\ js.\ jr\ <\ i \wedge js\ <\ jr$
$\mathit{received}_{mc}\ n\ tr\ jr \wedge \mathit{sent}_{mc}\ n\ tr\ js))$

Here $\mathit{sent}_{mc}\ m\ tr\ i$ is a property of MiniCaml traces, it says that message m was sent on tr at point i. Definition 2 says, if message m is being sent then all previous messages must have been sent and acknowledged.

The proof that all traces of the MiniCaml program *stenning_send* satisfies *thread_send_property* is not given here (similarly for *stenning_receive*). However, they are not not simply assumed. They were proved by deriving, in Isabelle, a small logic from the semantics of MiniCaml. The logic allows safety properties to be stated and proved for MiniCaml programs. The important part of the proof is discovering the correct loop invariant. For the sending program an appropriate invariant is that all messages less than `count` have been sent and acknowledgements for then have been received.

The proofs for the MiniCaml programs do not immediately apply in networks. They are properties of MiniCaml traces not network traces. As explained in Sections 2 and 3, these labels from the MiniCaml semantics and are lost by the synchronisation at the network level. So the fact that *thread_send_property* holds of *stenning_send* does not immediately apply to the network N_S.

Despite the labels not being present on a network trace, the MiniCaml steps must have taken place (the semantics in Figure 1 tells us that network steps involving threads also involve steps from the MiniCaml semantics). They will be represented by some network labels, tau (for internal actions), calls to the socket library etc. Thus there is a relationship between network traces and thread traces. Lemma 1 tells us about this relationship.

Lemma 1. *All network traces also yield traces for threads in the network, and each step of the thread trace must correspond to some step in the network trace.*

The proof is carried out by firstly showing the step property that *any single network step either leaves a thread unchanged or allows the thread to perform one step from the MiniCaml semantics.* Using this property in an induction over the definition of network traces gives the final result.

For some arbitrary network trace, we can not recover what the thread traces are, nor exactly which steps relate to which. But by capturing the relationship between the two types of traces, Lemma 1 allows properties of MiniCaml programs to be lifted into networks. Consider the network N_S we would like to show that this network has an equivalent property to *thread_send_property*, i.e. a property about UDP messages sent and received on the network. This proof will require two fundamental properties about hosts in networks.

Lemma 2. *Hosts do not receive invented messages.*

Lemma 3. *Hosts do not invent messages to send.*

Lemma 2 tells us that if a host places a message in the incoming queue of a socket then the message came from a UDP packet on the network. Lemma 3 says that if a host puts a UDP packet on the network then that packet must have come from some thread making a call to the socket library function `sendto()`. We would expect these properties to be true, proving them helps give assurance that our semantics is working correctly.

A large number of properties, which like Lemmas 2 and 3 describe how the semantics works, are required. For a small number I first assumed the properties as axioms and then filled in the proofs later. Only two of these assumptions have not yet been formally proved, and they are only required in one place (the final Theorem). I am currently completing the proofs of these final assumptions.

Using these meta-properties, a property equivalent to *thread_send_property* (the same property except that it is about UDP packets sent over and received from the network) can be proved for the network N_S.

The proof is by induction on the definition of network traces. We must consider the following cases
Base case: Trivial
Step case: We have to show for arbitrary m, i and l that if $\mathit{sent\ sender}\ m\ (tr@l)\ j$ then all previous messages have been sent and received. Clearly the case to consider is when $i = \mathit{length}(tr)$ (i.e. l is the sending step), other cases for i are covered by the inductive hypothesis. Using Lemma 3 we know that the message sent in this last step must have come at some point from a thread performing a send. With Lemma 1 we know that there must be a trace of *stenning_send* including this send. Further, the trace must satisfy *thread_send_property*, as all traces of *stenning_send* do. *thread_send_property* tells us that there are send and receive actions on the thread trace for previous messages. Now Lemmas 1, 2 and 3 help us to derive that corresponding messages must have been sent and received at the network level. Thus we have shown that the network N_S alway sends the messages in order.

A property asserting that messages are accepted in order can be proved in the same way for the *stenning_recv* program. In a similar way it can be lifted to a property of the traces of N_R.

The property in Definition 2 asserts that the program *stenning_send* is behaving correctly. Similarly the properties of N_S and N_R assert that each half of the network is well behaved. Now we need to combine these results to prove a final theorem about the interaction of these networks.

Theorem 1 (Safety).
$\forall\ tr\ N'.\ N_{initial} \overset{tr}{\Longrightarrow} N' \longrightarrow stenning\_property\ tr$

Proof. by arguments about the occurrence of events on traces. For an arbitrary m, that has just been accepted (*accepted receiver m tr i*), two things need to be shown.

- Message m was sent by the sender. As m has been received and accepted we know with Lemma 2 that it must have been sent across the network at some stage. It is easy to show that only the sender sends messages to the receiver, so message m must have been sent by the sender.
- All previous messages have already been accepted. This is an easy consequence of the property already proved for N_R, which states that messages are only ever accepted in order.

(*End of Proof*)

Theorem 1 with Definition 2 (lifted into networks) tell us that messages are sent out in order and accepted in order. Thus, in terms of this safety property, the sending an receiving MiniCaml programs will work correctly in a UDP network.

6 Conclusion

This paper has discussed the verification of UDP based distributed programs. A model of UDP and the sockets library was presented. Stenning's protocol was implemented in a fragment of OCaml. The verification of a safety property for the protocol was discussed. The entire development was carried out in the theorem prover Isabelle.

6.1 Related Work

Stenning's protocol and its close relatives, the alternating bit and sliding window protocols, have been scrutinised on a number of other occasions (Misra, Chandy & Smith 1982, Halpern & Zuck 1992, DiVito 1982). The work presented here seems to be the first proof at such a concrete level.

Lynch presents the protocol in I/O Automata, giving hand proof demonstrating that the protocol acts as a reliable FIFO channel (Lynch 1996). Schumann discusses the theorem prover SETHEO and its application to verification of the protocol (Schumann 1995).

Others have also considered the idea of verifying network code. Dam *et al.* developed a method for the verification of Erlang programs (Dam, Fredlund & Gurov 1998, Arts & Dam 1999), though they have a much simpler model of network communication.

6.2 Future Work

It is encouraging that we can carry out formal proof in the UDP model. It was originally designed as a post hoc specification of the sockets library and Internet communication. Following the success of this verification the next milestone is to implement a larger distributed protocol in the system (possibly one relying on host crashes, disconnection and message loss) and attempting to prove challenging properties, such as those involving liveness or timing constraints.

This proof could also be extended. The program could be extended to act as a library function that passes actual messages. The proof could then be extended to arbitrary networks, rather than the simple two host networks considered here. In this way I might be able to use this protocol as the underlying communication mechanism for a larger protocol.

I am also interested in deriving better techniques for compositional proof in large models such as this one. Compositional techniques that make use of predicate transformers and semantically characterise predicates may be helpful in this task (Charpentier & Chandy 2004).

7 Acknowledgements

Financial support was provided by the Computer Laboratory at the University of Cambridge, the Cambridge Commonwealth Trust, and CSIRO Australia.

References

Arts, T. & Dam, M. (1999), Verifying a distributed database lookup manager written in Erlang, *in* 'FM'99—Formal Methods, Volume I', Vol. 1708 of *Lecture Notes in Computer Science*, Springer, pp. 682–700.

Bhargavan, K., Obradovic, D. & Gunter, C. A. (2002), 'Formal verification of standards for distance vector routing protocols', *Journal of the ACM (JACM)* **49**(4), 538–576.

Charpentier, M. & Chandy, K. M. (2004), 'Specification transformers: A predicate transformer approach to composition', *Acta Informatica* **40**(3), 265–301.

Chkliaev, D., Hooman, J. & Vink, E. d. (2003), Verification and improvement of the sliding window protocol, *in* H. Garavel & J. Hatcliff, eds, 'Proc. TACAS 2003', LNCS, p. 15pp.

Dam, M., Fredlund, L. & Gurov, D. (1998), 'Toward parametric verification of open distributed systems', In *Compositionality: the Significant Difference,* H. Langmaack, A. Pnueli and W.-P. de Roever (eds.), Springer **1536**, 150–185.

de Roever, W.-P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M. & Zwiers, J. (2001), *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, number 54 *in* 'Cambridge Tracts in Theoretical Computer Science', Cambridge University Press, Cambridge, UK.

DiVito, B. (1982), Verification of the stenning protocol, Technical report, University of Texas at Austin.

Gordon, M. J. C. & Melham, T. F. (1993), *Introduction to HOL (A theorem-proving environment for higher order logic)*, Cambridge University Press.

Gordon, M., Milner, R. & Wadsworth, C. (1979), *Edinburgh LCF*, Vol. 78 of *Lecture Notes in Computer Science*, Springer.

Halpern, J. Y. & Zuck, L. D. (1992), 'A little knowledge goes a long way: Knowledge-Based derivations and correctness proofs for a family of protocols', *Journal of the ACM* **39**(3), 449–478.

Lynch, N. A. (1996), *Distributed Algorithms*, Morgan Kaufmann Publishers.

Milner, R. (1999), *Communicating and Mobile Systems: the Pi-Calculus*, Cambridge University Press.

Misra, J., Chandy, K. M. & Smith, T. (1982), Proving safety and liveness of communicating processes with examples, *in* 'Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing', ACM Press, pp. 201–208.

Nipkow, T., Paulson, L. C. & Wenzel, M. (2002), *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *LNCS*, Springer.

Nipkow, T., Paulson, L. & Wenzel, M. (2003), *Isabelle's logics: HOL.* `http://isabelle.in.tum.de/dist/Isabelle2003/doc/logics-HOL.pdf`.

Postel, J. (1980), 'User Datagram Protocol, STD 6, RFC 768', Internet Engineering Task Force. `http://www.ietf.org/rfc.html`.

Postel, J. (1981), 'Internet Protocol, STD 5, RFC 791', Internet Engineering Task Force. `http://www.ietf.org/rfc.html`.

Schumann, J. (1995), Using the theorem prover SETHEO for verifying the development of a communication protocol in FOCUS: A case study, *in* 'Analytic Tableaux and Related Methods', pp. 338–352.
*citeseer.ist.psu.edu/schumann95using.html

Serjantov, A., Sewell, P. & Wansbrough, K. (2001*a*), The UDP calculus: Rigorous semantics for real networking, *in* 'Proceedings of TACS 2001: Theoretical Aspects of Computer Software (Sendai), LNCS 2215', pp. 535–559.

Serjantov, A., Sewell, P. & Wansbrough, K. (2001*b*), The UDP calculus: Rigorous semantics for real networking, Technical Report 515, Computer Laboratory, University of Cambridge. `http://www.cl.cam.ac.uk/users/pes20/Netsem/`.

Simons, D. P. L. & Stoelinga, M. (2001), 'Mechanical verification of the IEEE 1394a root contention protocol using uppaal2k', *International Journal on Software Tools for Technology Transfer* **3**(4), 469–485.

Stevens, W. R. (1998), *UNIX Network Programming Vol. 1: Networking APIs: Sockets and XTI*, second edn, Prentice Hall.

Stoelinga, M. & Vaandrager, F. (1999), 'Root contention in IEEE 1394', *Lecture Notes in Computer Science* **1601**, 53–74.

Wansbrough, K., Norrish, M., Sewell, P. & Serjantov, A. (2002), Timing UDP: Mechanized semantics for sockets, threads, and failures, *in* 'Proceedings of ESOP 2002: the 11th European Symposium on Programming (Grenoble), LNCS 2305', pp. 278–294.

Xavier Leroy et al (2004), *The Objective-Caml System, Release 3.08*, INRIA. `http://caml.inria.fr/ocaml/`.