Rigour is good for you *and* feasible: reflections on formal treatments of C and UDP sockets

Michael Norrish Peter Sewell Keith Wansbrough
{Firstname.Lastname}@cl.cam.ac.uk
Computer Laboratory, University of Cambridge, Cambridge CB3 0FD

1. Introduction

We summarise two projects that formalised complex real world systems: UDP and its sockets API, and the C programming language. We describe their goals and the techniques used in both. We conclude by discussing how such techniques might be applied to other system software and by describing the benefits this may bring.

2. Specifying UDP and the sockets API

We recently formalised a substantial behavioural specification, that for the Internet protocol UDP, as presented to programmers through the *sockets interface* [12, 10, 1, 5, 11]. Our aim was to make clear the behavioural subtleties of the widely used – but poorly documented – sockets API. This clarification of the interface should ease the production of robust software that uses it. The specification was necessarily developed *post hoc*; we developed it by referring to existing documentation (RFCs and source code, for example), and by experimentally checking existing implementations, using automated tools. We produced the specification in the following three stages:

Syntax We developed a precise notation for describing our systems. We specified abstract syntax with which to denote values corresponding to threads, hosts, messages, sockets, an abstract network, and a simple functional language in which to write programs for threads. Specifying syntax can be seen either as writing a grammar (using BNF notation, say), or as writing data type descriptions in a programming language, particularly one with convenient support for expressing disjoint unions and recursion.

For example, using an ML-like notation, the body of an IP message is specified as in Figure 1. The first possibility of the three above corresponds to a UDP message, with optional destination and source ports, as well as a message body. The remaining two possibilities correspond to the two types of ICMP message we modelled, unreachable host and port messages respectively, each with source and destination details recorded. This is an abstraction of the structure of actual IP packets: there are many other fields, but they are not significant for programs that use the part of the sockets interface we consider, so need not be modelled.

Typing We developed a set of typing rules, expressing constraints on the values generable from the abstract syntax. For example, our network hosts include a list of active sockets and we require that each of these have a unique file descriptor. Another constraint is that no message in a socket's incoming queue should include a "martian" IP address. These constraints are both examples of *assertions*, requirements that might be checked dynamically. We also specified a more traditional type system for our miniature programming language.

Behaviour Finally, we described the legal behaviours of the system. Using a labelled transition system (a particular form of automata), we specified the possible ways in which the system might evolve. Our model encompasses threads in hosts, the various non-deterministic behaviours assumed of networks (message dropping, duplication and misordering) and timing requirements.

We illustrate with two examples from the 78 rules that describe the interactions between network hosts, the network and the threads of a process running on a host. Each rule is of the general form

$$h_0 \xrightarrow{label} h$$

Here h_0 is the state of a network host's kernel, and $label$ describes an interaction between that kernel and either the network or the threads of a user-level process. The result of this interaction is presented in the expression h.

In Figure 2, our first rule describes a thread with identifier tid `connect`-ing a socket to an external address. The evaluation context $\mathcal{F}$ picks out the pertinent components of

```
ipBody = UDP of (port option * port option * string)
       | ICMP_HOST_UNRCH of (ip * port option * ip * port option)
       | ICMP_PORT_UNRCH of (ip * port option * ip * port option)
```

Figure 1. Type declaration for the body of an IP message

the given host. The $ifds$ component is the host's network interfaces. The next two components specify that in the state before the interaction, the host records that thread tid is running. After the interaction (thread tid making its call), the host records that the host is due to return a unit value (representing success) to the same thread. In the meantime, that thread is blocked waiting for this return to happen.

The main effect of the call is to alter the state of the socket with the file descriptor fd, which is the last component of the tuple. It has its destination IP address and destination port (fourth and fifth fields within the SOCK tuple) modified to take on the values of the i and ps parameters. The source address and port also change, with the source port becoming "autobound" if it is not bound already, and the source address being generated by examining the host's current interfaces ($ifds$) and the destination.

Our second example, in Figure 3, illustrates a host receiving a packet off the network and delivering to a matching socket. The context $\mathcal{S}$ is an injective function that inserts a socket (here s, initially) into an implicit list of other sockets. The rule describes a transition where one of the host's sockets changes, but where the host's other sockets, and its other components all remain unchanged. The label on the transition arrow is the incoming packet. The lookup function returns the set of scokets that such a packet could be delivered to, and the other side conditions check that the IP addresses involved are reasonable. The resulting state is updated so that the message queue of socket s is extended with the message, paired with interface information about how the message was received.

3. Specifying C

A similar approach was taken in the formalisation of the semantics of C done by the first author [7, 8]. This project was undertaken with the aim of showing that the tools of the theoretical community could be used to model a real world programming language, and do so with high confidence in the correctness of the result.

Syntax The concrete syntax of C expressions and statements is specified in the ISO Standard [6]. Generating an abstract syntax that elided details only necessary for parsing was trivial.

Typing The type system in the standard is carefully described in natural language. The formalised system is non-trivial only in its slightly complicated treatment of l-values and normal values. For example, the expression `e.fld1`, with the field `fld1` declared as an array, can only be a normal value if the expression e is an l-value. There is also a lot of tedious detail in specifying parts of the system like the *usual arithmetic conversions*, which determine the types of the results of binary arithmetic operators.

Behaviour This project did not specify the behaviour of the standard's library. A great deal of complexity remains in the core language, however. In particular, the details of C's sequence points, and the degree to which expression evaluation is allowed to refer to and update objects in memory, are very complicated. These and the standard's three forms of under-determinism, *implementation-defined*, *unspecified* and *undefined* behaviours, were modelled very carefully.

The operational rules used have conclusions of the general form

$$\langle e_0, \sigma_0 \rangle \rightarrow \langle e, \sigma \rangle$$

Here an expression e_0, coupled with a state σ_0, reduces to a new expression-state pair. States include, among other components, the contents of memory, which parts of memory are allocated, and which parts are initialised.

For example, Figure 4 presents two rules for expressions involving binary operators other than `&&`, `||` and `,` (comma). The first rule can be paraphrased: if e_1 can take a step to e_1' (transforming state σ_0 to σ on the way), then the expression $e_1 \oplus e_2$ can reduce to $e_1' \oplus e_2$. When the arguments to the operator $\oplus$ have been fully evaluated, the operator will take effect, reducing $value_1 \oplus value_2$ to the appropriate result.

By way of contrast, there is one rule for `&&`, allowing only the first argument to be evaluated, which enforces that operator's "short-circuit" behaviour:

$$\frac{\langle e_1, \sigma_0 \rangle \rightarrow \langle e_1', \sigma \rangle}{\langle e_1 \texttt{ \&\& } e_2, \sigma_0 \rangle \rightarrow \langle e_1' \texttt{ \&\& } e_2, \sigma \rangle}$$

C's rules governing the application of side effects are similarly under-determined. Side effects need not be applied as they are generated, nor in order. This under-determinism is tempered by strong constraints, which deem certain sequences of memory references and updates to result in *undefined behaviour*. Programs that might exhibit such sequences are in the same class as those that divide by zero, or access uninitialised memory.

$$\mathcal{F}(ifds, tid, \text{RUN}_d, \text{SOCK}(fd, \texttt{*}, ps_1, \texttt{*}, \texttt{*}, e, f, mq))$$

$$\xrightarrow{tid \cdot \texttt{connect(}fd, i, ps\texttt{)}}$$

$$\mathcal{F}(ifds, tid, \text{RET}(\text{OK }())_{dsch}, \text{SOCK}(fd, \uparrow i_1, \uparrow p'_1, \uparrow i, ps, e, f, mq))$$

where $\mathsf{F\_context}(\mathcal{F}) \ \wedge \ i_1 \in \mathsf{outroute}(ifds, i) \ \wedge \ p'_1 \in \mathsf{autobind}(ps_1, \mathcal{F})$

Figure 2. Performing a `connect` call

h with sks := $\mathcal{S}(s)$

$$\xrightarrow{\texttt{IP}(i_3, i_4, \mathsf{UDP}(ps_3, ps_4, data))}$$

h with sks := $\mathcal{S}(s$ with mq := APPEND $s.mq$ $[(\texttt{IP}(i_3, i_4, \mathsf{UDP}(ps_3, ps_4, data)), ifd.ifid)])$

where $\mathsf{socklist\_context}(\mathcal{S}) \wedge s \in \mathsf{lookup}(\mathcal{S}(s), (i_3, ps_3, i_4, ps_4)) \wedge$
$ifd \in h.ifds \wedge i_4 \in ifd.ipset \wedge i_4 \notin \mathsf{LOOPBACK} \wedge$
$i_3 \notin \mathsf{MARTIAN} \cup \mathsf{LOOPBACK}$

Figure 3. Receiving a packet from the network

$$\frac{\langle e_1, \sigma_0\rangle \rightarrow \langle e'_1, \sigma\rangle}{\langle e_1 \oplus e_2, \sigma_0\rangle \rightarrow \langle e'_1 \oplus e_2, \sigma\rangle} \qquad \frac{\langle e_2, \sigma_0\rangle \rightarrow \langle e'_2, \sigma\rangle}{\langle e_1 \oplus e_2, \sigma_0\rangle \rightarrow \langle e_1 \oplus e'_2, \sigma\rangle}$$

Figure 4. Behaviour of C's binary expressions

4. Reflections

Both projects described above were examples of *post hoc* specification. Our specifications required us to compare our developing models with real world, un-formalised, systems. In the case of C, the un-formalised system was the ISO standard. Being quite carefully written, checking the formal specification manually was not infeasible. In addition, the formalisation was also checked by the proof of a great variety of additional theorems. These were not necessarily particularly deep results, indeed were often quite trivial. When these proofs failed, this pointed out a problem, either in the model, or occasionally, in expectations of how the model should behave.

In the case of UDP sockets, there is no such natural-language standard. The RFCs are not complete, and in any case specify only the wire behaviour not the sockets API behaviour. In addition to consulting documentation, we used experimental techniques to determine the behaviour of implementations in particular situations. These experiments were important in determining behaviours in extreme cases that were often not very well documented.

In both cases, we also used the HOL interactive theorem-proving system [3, 9]. For the C semantics, HOL was used from the outset, enabling the definition of a type representing C syntax, and the appropriate relations, for typing and behaviour, over those types. Used entirely in this "definitional capacity", HOL provides an expressive, statically type-checked language in which to write logical definitions. HOL's theorem-proving facilities were subsequently used to prove the theorems referred to above.

With UDP, we found HOL's greatest benefit to be the sanity checks we were able to make of the specification. The mechanisation of what was originally a pen-and-paper specification caught numerous errors. Most of these were typographical errors, such as using functions or predicates with inappropriate types or numbers of arguments. Nonetheless, we also later proved that our behavioural system preserved an important invariant (or safety) property. Attempting this proof caught a number of omissions and errors in the original model, drawing our attention to its obscure corners.

We have yet to marry our specifications of UDP and C to verification, in any significant way. Nonetheless, our specifications are valuable in themselves. We have given formal description to large and complicated systems, and in so doing, have made explicit the contract between users and implementors of those systems. Our (annotated!) specifications provide detailed documentation which may be useful to both users and implementors of UDP sockets and C compilers. For UDP, and with mechanical assistance, we have explored both the state spaces of the formal system, and also of the deployed implementations, discovering, for example, how the Windows and Linux implementations differ.

This is a general phenomenon: while underway, the specification activity brings benefits of increased understanding. Later, the specification is useful as precise documentation. Subsequent program verification, while desirable, remains infeasible in most contexts.

5. Lessons—what others can do

Language designers give rigorous specifications of language syntax as a matter of course. Moreover, there are long-established formal notations for the specification of concrete syntax, and tools to support the use of these notations. Syntax is specified not just for programming langauges, but other formal languages such as HTML and XML.

Rigorous definitions of type systems is less common, though standard practice in the programming language and semantics communities. These definitions are usually expressed in informal mathematics, using a body of widely-accepted stylised idioms, rather than in any particular logical system. General tool support is lacking, though interactive proof assistants such as HOL are increasingly used in large-scale work. In addition to the often elaborate type systems constructed around research calculi and languages, types have been usefully deployed in, for example, the Xduce project [4], which defines regular expression types for manipulating XML data, and the Hafnium project [2] which used type inference to address the Y2K problem for COBOL programs.

The situation for behavioural specification is even worse. A very weak form is the use of assertions, becoming more widespread. "State-of-the-art" in most fields is more-or-less precise natural language. Network protocols at lower levels of abstraction are specified precisely, in RFCs and other documentation, and some hardware components do have formal specifications, but we are unaware of significant examples at the systems level.

We do not propose a universal methodology for writing formal specifications to address these problems. Indeed, we believe that no such methodology is pragmatically viable. Our experience suggests, however, that a number of important principles apply:

Specify in small pieces. Do not set out to specify everything all at once, nor in exhaustive detail. Attack tractable pieces of systems, at appropriate levels of detail, and begin with simple aspects such as typing requirements.

Choose the right pieces. Formal specification is most valuable for systems with subtle behaviour, for which it is hard to develop a sufficiently good informal understanding, and (obviously) for systems for which cor-

rectness matters. Concurrent systems, including communication and security protocols, often benefit.

Use intellectual tools. Draw on techniques from appropriate fields, such as operational semantics, type theory, programming language theory, and concurrency theory. Take an existing specification language/idiom or build a new one if required, as appropriate to the problem. Make sure, however, that whatever specification notation is used has a precise meaning.

Use mechanical tools. Use a notation with tool support, so that mechanical analysis and checking of the specification is possible. This might be simple type-checking, model-checking, machine-assisted theorem proving, or automated testing.

Begin as early as possible. *Post hoc* specification activities such as the two we described here have value, but if used at the design stage then rigorous treatments can be valuable in other ways, providing early understanding and a push towards clean design.

In the long term, we believe greater rigour is essential in the development of more robust software at all levels; there are many behaviourally subtle aspects of operating systems which could benefit from it, and for which the tools are now available.

References

[1] U. CSRG. 4.2BSD, 1983.

[2] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sørensen, and M. Tofte. AnnoDomini: from type theory to year 2000 conversion tool. In *POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

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

[4] H. Hasoya and B. C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, 2001.

[5] IEEE. *Information Technology—Portable Operating System Interface (POSIX)—Part xx: Protocol Independent Interfaces (PII), P1003.1g*. Mar. 2000.

[6] *Programming languages – C*, 1990. ISO/IEC 9899:1990.

[7] M. Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.

[8] M. Norrish. Deterministic expressions in C. In S. D. Swierstra, editor, *Programming languages and systems, 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 147–161. Springer, March 1999.

[9] M. Norrish and K. Slind. A thread of HOL development. *Computer Journal*, 45(1):37–45, 2002.

[10] A. Serjantov, P. Sewell, and K. Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proceedings of TACS 2001: Theoretical Aspects of Computer Software (Sendai, Japan), LNCS 2215*, pages 535–559, Oct. 2001.

[11] W. R. Stevens. *UNIX Network Programming Vol. 1: Networking APIs: Sockets and XTI*. Prentice Hall, second edition, 1998.

[12] K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: Mechanized semantics for sockets, threads and failures. In *Programming languages and systems, 11th European Symposium on Programming*, Lecture Notes in Computer Science. Springer, 2002. To appear.