Mathematical rigour, pragmatically:
the behaviour of C and UDP

Michael Norrish, Peter Sewell
and Keith Wansbrough

UNIVERSITY OF CAMBRIDGE
Computer Laboratory
Motivation

- Work stemmed from desire to attack real world problems.
- We believe that more rigour would be helpful...
- ...so try it and see (exercising various theoretical techniques).
- Not on whole OS’s, but not toy problems either.
- Spent some time; didn’t hate it too much; even half enjoyed it.
- Think that rigour is doable, and “good for you” too.

- Demonstration today of what, how, and why.
Comparison of sources

Both *post hoc*.

**UDP:**
- Used RFCs, OS documentation, Linux/BSD source code
- Clarified with experimental validation

**C:**
- ISO standard (C90)
- Consultation with others (e.g., `comp.std.c`) clarified ambiguities
UDP—Motivation: The Semantic Gap

**Thesis:** Complexity makes it hard to understand the behaviour of distributed systems (formally or informally) based only on informal descriptions.
UDP—Motivation

We want to be able to:

- reason about distributed programs,
- written in general-purpose programming languages,
- using standard communication primitives,
- in the presence of failure and disconnection.

We chose to examine UDP/ICMP and the Sockets API:

- real-world (and ubiquitous)
- simple failure models
Networks and Protocols—Abstraction

192.168.0.11 192.168.0.21 192.168.0.12 192.168.0.1

IP(192.168.0.11,192.168.0.14,UDP(..))

IP(192.168.0.14,192.168.0.11,ICMP-PORT-UNREACH(..))

23 September 2002
Networks and Protocols—Syntax

IP addresses $i$: 32-bit values, eg 192.168.0.11.

IP datagrams $ip ::= IP(i_1, i_2, body)$

UDP ports $ps ::= * | 1 | \ldots | 65535$

UDP and ICMP datagrams are IP datagrams with bodies

\[
\text{body ::= } \\
\text{UDP}(ps_1, ps_2, data) \\
\text{ICMP\_PORT\_UNREACH}(is_3, ps_3, is_4, ps_4) \\
\text{ICMP\_HOST\_UNREACH}(is_3, ps_3, is_4, ps_4).
\]
The Sockets API

The sockets interface

**socket** : () → fd
**bind** : fd * ip * port↑ → ()
**connect** : fd * ip * port↑ → ()
**disconnect** : fd → ()
**getsockname** : fd → ip↑ * port↑
**getpeermaname** : fd → ip↑ * port↑
**sendto** : fd * (ip * port↑)↑ * string * bool → ()
**recvfrom** : fd * bool → ip * port↑ * string
**geterr** : fd → error↑
**getsockopt** : fd *sockopt → bool
**setsockopt** : fd *sockopt * bool → ()
**close** : fd → ()
**select** : fd list * fd list * int↑ → fd list * fd list
**getifaddrs** : () → (ifid * ip * ip list * netmask) list

**port_of_int** : int → port
**ip_of_string** : string → ip
**UDP** : error → exn

Thread operations

**create** : (T → T') → T → tid
**delay** : int → ()

Basic operating system operations

**print_endline_flush** : string → ()
**exit** : () → void
UDP Sockets: Things We Have To Pay Attention To

- irregular use of IP and port wildcards
- many local errors e.g., `bind`: port in use, port in privileged range, IP not one of this machine, OS run out resources, `fd` not a socket
- machines have multiple IP addresses, and multiple interfaces
- asynchrony; blocking calls (`sendto, recvfrom, select`)
- message reordering, loss and duplication
- host failure and disconnection/reconnection
- `ICMP_PORT_UNREACH` generation and socket error flags

Focussing especially on the *information about failure* that is visible through the sockets interface.
Sockets and Hosts—Syntax

The main host component is the OS state:

\[ h ::= \text{HOST}(\text{conn}, (\text{ifds}, \text{ts}, s, oq, oqf)) \]

in which each communication endpoint is represented by a socket:

\[ \text{SOCK}(fd, is_1, ps_1, is_2, ps_2, es, f, mq) \]
UDP Invariants (Typing)

Invariants include:

- The file descriptor associated with a socket in a host should be associated only with that socket.
- No message in a socket’s incoming queue should include a “martian” address.
- If a thread is blocked on a `sendto` system call to descriptor `fd`, then the host should include a socket with descriptor `fd`, and that socket should have its source port bound.

And many (more complicated) others...
UDP Behaviour

Express behaviour as labelled transition systems (automata) of a particular form.

The main definition is the semantics of hosts:

\[ h \xrightarrow{\ell} h' \]

defined by axioms – for each socket call and for sending/receiving messages to the network.
UDP—Example Host Rule

\[ \text{sendto}_1 \quad \text{succeed} \quad \text{autobinding} \]

\[ h \text{ with } (ts := ts \oplus (\text{tid} \mapsto (\text{RUN})_d)); \]
\[ s := \text{SC}(s \text{ with } es := \ast)) \]

\[ \text{tid} \cdot \text{sendto}(s.f, ips, data, nb) \]

\[ h \text{ with } (ts := ts \oplus (\text{tid} \mapsto (\text{RET(OK())})_d)); \]
\[ s := \text{SC}(s \text{ with } (es := \ast; ps_1 := \uparrow p_1)); \]
\[ oq := oq'; oqf := oqf' \]

socklist-context \( SC \land p_1' \in \text{autobind}(s.ps_1, SC) \land \)

string-size \( data \leq \text{UDPpayloadMax} \land ((ips \neq \ast) \lor (s.is_2 \neq \ast)) \land \)

\( (oq', oqf', \top) \in \text{dosend}(h.ifds, (ips, data), (s.is_1, \uparrow p_1', s.is_2, s.ps_2), h.oq, h.oqf) \)
C—Motivation

How hard can real, formal software verification be, anyway?

Later: the researcher as intrepid taxonomist.

A combination of

- almost 20 years in the wild
- standardisation
- use in widely different contexts (applications to operating systems to device drivers)

has produced an interesting monster.
C—Abstraction

What to leave out:

- the library (system calls etc)
- unions
- `goto` & `switch`
- bit-fields

What to retain:

- the rest of the language
- under-specification
- ISO Standard’s virtual machine

Focus on compiler and architecture independence: the purist’s *strictly conforming* C.
C—Syntax

For example, C’s types:

$$\tau ::= \text{int} | \text{char} | \ldots | \tau^* |$$
$$\tau[n] | \tau^* \rightarrow \tau |$$
$$\text{struct} \ tag$$

(Not all possibilities are valid types: must forbid arrays of zero size; functions returning arrays . . . )

Similar definitions for expressions and statements.
C—Typing

Rules for address-taking and pointer dereference:

\[
\frac{\Gamma \vdash e : \text{obj}[\tau]}{\Gamma \vdash \&e : \tau^*} \quad \frac{\Gamma \vdash e : \tau^* \quad \tau \neq \text{void}}{\Gamma \vdash *e : \text{obj}[\tau]}
\]

The type \text{obj}[\tau] is an l-value of type \tau.

Variables also have \text{obj}[\tau] type.
C—Three forms of under-specification

- **Implementation defined:** e.g., number of bits in a byte
- **Unspecified:** e.g., order of evaluation of arguments to binary arithmetic operators
- **Undefined:** illegal behaviours:
  - running off the end of arrays
  - accessing uninitialised memory
  - casting values to incompatible types
  - dividing by zero

*Implementations* may do Weird Stuff when these things happen; the semantics regards them all as aborts.
C—Unspecified vs. Undefined

Side effects are unspecified, in that

- Side effects need not be applied immediately
- Side effects need not be applied in order

So, with $v$ initially 3,

\[ v++ + v++ + v++ + v++ \]

might result in values anywhere between 12 and 18. (Mightn’t it?)
C—More Undefined Behaviour

Actually,

\[ v++ + v++ + v++ + v++ \]

is undefined because...

...within a “phase” of expression evaluation,

- updating the same object twice is undefined behaviour
- updating and referring to the same object is undefined behaviour, unless the reference was made to calculate the new value
## C—Undefinedness Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>v++ + v++</td>
<td>Undefined</td>
</tr>
<tr>
<td>v + v++</td>
<td>Undefined</td>
</tr>
<tr>
<td>v++ + *i</td>
<td>Undefined*</td>
</tr>
<tr>
<td>v = v + 1</td>
<td>OK†</td>
</tr>
<tr>
<td>a[a[i]] = 0</td>
<td>?</td>
</tr>
</tbody>
</table>

(*) if `i` points to `v`

(†) “updating and referring to the same object is undefined behaviour, unless the reference was made to calculate the new value”

(?) if `a[i] == i`
Feasible—how did we do these things?

An *ad hoc* collection of techniques. No One True Way.

- **Mathematical techniques:** timed operational semantics automata for the components of hosts (OS, shared memory, threads), synchronisation techniques, programming language semantics.

- **Software tools:**
  - HOL (type-checking, proving sanity properties)
  - automated testing
  - OCaml sockets and threads libraries
  - automated typesetting

- **Time:** C and UDP both roughly 2 person years.
Good for you?—The *post hoc* story

- **Documentation**: Formal specifications make natural language precise and unambiguous (sanity checking)
- **Meta-theorems**: Proofs of meta-theorems become possible
- **Machine Processable**: The basis for our typesetting code and other potential applications
- **Education**: Formal specification forces the specifier to understand the object of study

Our work is pragmatic. It’s based on

- choosing the rights things to formalise
- testing of specifications as they are developed
- experimentation with real code
What about verification?

- There is no silver bullet
- No one specification methodology is right for all cases
- Getting a specification right can provide most value
- Software Verification technology still makes users’ lives miserable.
Good for you?—The *pre hoc* story

You can derive considerable benefit by expressing designs rigorously from the outset. Recent examples:

- Microsoft’s IL (Intermediate Language) for .NET
- Cyclone, C-- (modern low-level languages)
- Protocols (including security)
Conclusion & Future Work

Rigorous description of the behaviour of real systems is feasible.

It can be a valuable tool for documentation (post-hoc) and design (pre-hoc).

...though one must take care to choose the right pieces to specify, and use appropriate intellectual and mechanical tools.

For the future

- have started on TCP (yuk)
- design new, high-level distributed layers on sound foundations
- redesign the world :-(

23 September 2002