Formal Co-Validation of Low-Level Hardware/Software Interfaces

Alex Horn\textsuperscript{1}  Michael Tautschnig\textsuperscript{1}  Celina Val\textsuperscript{2}  Lihao Liang\textsuperscript{1}  
Tom Melham\textsuperscript{1}  Jim Grundy\textsuperscript{3}  Daniel Kroening\textsuperscript{1}

\textsuperscript{1}University of Oxford
\textsuperscript{2}University of British Columbia
\textsuperscript{3}Intel Corporation

October 22, 2013
Motivation

Consider this scenario:

• The product won’t function unless there is firmware! So ideally ...
• But the hardware won’t be available until shortly before release.

Our focus: how can we formalize hardware/software interfaces?
Current techniques

Well-known firmware development techniques in industry include:

- Using an older version of the hardware (if any!)
  - hard to debug, can hide latent firmware bugs, no guarantee
- Virtual platforms
  - faster turnaround times, easier to debug and test
  - but generally too big to formally analyze

Idea: Verifiable Virtual Platform

How to model hardware/software interfaces so existing software engineering principles apply but also formal methods

(See also question posed by Per Bjesse (Synopsys) during FMCAD 2010)
Current techniques

Well-known firmware development techniques in industry include:

- Using an older version of the hardware (if any!)
  - hard to debug, can hide latent firmware bugs, no guarantee
- Virtual platforms
  - faster turnaround times, easier to debug and test
  - but generally too big to formally analyze

Idea: Verifiable Virtual Platform
Current techniques

Well-known firmware development techniques in industry include:

• Using an older version of the hardware (if any!)
  ◦ hard to debug, can hide latent firmware bugs, no guarantee
• Virtual platforms
  ◦ faster turnaround times, easier to debug and test
  ◦ but generally too big to formally analyze

Idea: Verifiable Virtual Platform

How to model hardware/software interfaces so existing software engineering principles apply but also formal methods

(See also question posed by Per Bjesse (Synopsys) during FMCAD 2010)
VVP: Verifiable Virtual Platform

realistic • open source • concurrent

Hardware/Software Interface

Low-level Software from GNU/Linux

QEMU Hardware Model in C

Software threads model untimed delays
Outline

An Ethernet MAC concurrency bug
   A real bug firmware developers care about

Problem statement and contribution

Technical details
   Few glimpses at a verifiable virtual platform

Experiments and conclusion
   Download Me!
GNU/Linux + Open Cores Ethernet MAC

Explain known kernel bug due to concurrency (i.e. asynchronous operations) in the hardware/software interface.
Concurrency bug

Interrupt source: \(0_a\)  

Interrupt mode? \(0_x\)

Buffer descriptors:

Initially, assume the firmware is in polling mode (i.e. \(0_x\)) and “there are no new RX frames” (yet).
Concurrency bug

Interrupt source: \(1_b\)  
Interrupt mode? \(0_x\)

Buffer descriptors:

A new RX frame arrives changing the interrupt source from 0\(_a\) to 1\(_b\). The arrival of an RX frame gives us a “nonempty” buffer descriptor.
Concurrency bug

Interrupt source: 0x1c

Interrupt mode?: 0x0

Buffer descriptors:

Repeat but notice that the Open Cores Ethernet MAC always sets the interrupt source register as new RX frames arrive (1b has become 1c).
Concurrenty bug

Interrupt source: \(1c\)  
Interrupt mode? \(0x\)

Buffer descriptors:

\[
\begin{array}{cccccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

The firmware reads one “nonempty” buffer descriptor changing it to be “empty” again.
Concurrent bug

Interrupt source: \[1_d\] Interrupt mode: \[0_x\]

Buffer descriptors:

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

But simultaneously new RX frames can arrive.
Concurrency bug

Interrupt source:

\[ 1_d \]

Interrupt mode?

\[ 0_x \]

Buffer descriptors:

As before, the firmware continues to consume these ...
**Concurrency bug**

Interrupt source: \(1_d\)  

Interrupt mode? \(0_x\)

Buffer descriptors:

... until it detects that there aren’t any more RX frames to consume. So assume it initiates a procedure now to switch to interrupt mode.
ConcURRENCY bug

Interrupt source:

\[ 1_e \]

Interrupt mode?

\[ 0_x \]

Buffer descriptors:

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Asynchronous operation: a fraction of a second later a new RX frame arrives and changes \(1_d\) to \(1_e\) as well as the buffer descriptor.
Concurrence bug

Interrupt source:  Interrupt mode?

$0_f$  $0_x$

Buffer descriptors:

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Since firmware is not in interrupt mode yet, it fails to detect the intermittent RX frame; it continues by clearing interrupt sources ($0_f$).
Concurrency bug

Interrupt source: $0_f$

Interrupt mode? $1_y$

Buffer descriptors:

```
 0 0 0 1 0 0 0 0 0
```

The firmware continues by enabling interrupts ($1_y$). But an interrupt is only raised once another RX frame arrives, *problem*. 
Polling to interrupt mode switch (bug)

Firmware

Ethernet MAC

Wire

New RX frame?

No

Clear interrupt source!

Enable interrupt mode!

New RX frame!
Polling to interrupt mode switch (fix)

Firmware

New RX frame?

No

Clear interrupt source!

New RX frame?

Yes

Stay in polling mode

Ethernet MAC

New RX frame!
Problem statement and contribution

Problem:

- Many firmware bugs can go undetected when hardware and software are verified in isolation.

Contribution:

- Three realistic and open-source benchmarks to scientifically study firmware verification.
- Practical evidence that a verifiable virtual platform is a feasible concept to verify hardware/software interfaces.
Benchmarks to study firmware verification

MC146818A: Real-time clock

TMP105: Temperature Sensor

Ethernet MAC

QEMU: open source processor emulator
Experimental setup

Overview of work flow:

1. Extract QEMU hardware model and Linux driver
2. Manually add runtime assertions in C
3. If necessary, introduce concurrency in QEMU + Linux code
   ◦ Use new CBMC concurrency source code annotations
   ◦ Encode any concurrency as symbolic partial orders (CAV’13)
4. SAT solver finds satisfying assignment (i.e. bug) or not.
Real-time clock (RTC)

Special-purpose registers that require an ancillary manipulation of bits to read and write time, date and alarm data.
## RTC benchmark code

<table>
<thead>
<tr>
<th>Project</th>
<th>Files</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Kernel 3.6</td>
<td>~ 14,000 (.h)</td>
<td>~ 17,000 (.c)</td>
</tr>
<tr>
<td>QEMU 1.2</td>
<td>~ 600 (.h)</td>
<td>~ 1,500 (.c)</td>
</tr>
<tr>
<td>QEMU hardware model of RTC</td>
<td>5 (.h)</td>
<td>5 (.c)</td>
</tr>
<tr>
<td>Linux x86 RTC driver and model</td>
<td>~ 300 (.h)</td>
<td>~ 8 (.c)</td>
</tr>
<tr>
<td>Combined RTC benchmark</td>
<td>0 (.h)</td>
<td>1 (.c)</td>
</tr>
</tbody>
</table>
Example Assertion

```c
void cmos_ioport_write(void *opaque, uint32_t addr, uint32_t data)
{
    RTCState *s = opaque;
    if ((addr & 1) == 0) {
        s->io_info = OUTB_0x70;  // for temporal property
        s->cmos_index = data & 0x7f;
    } else {
        switch(s->cmos_index) {
        case RTC_SECONDS_ALARM:
```

```c
    #ifdef RTC_BENCHMARK_PROP_9
    assert((s->cmos_data[RTC_REG_B] & REG_B_SET) == REG_B_SET);
    #endif
```

```c
    ...```
Example Assertion

```c
void cmos_ioport_write(void *opaque, uint32_t addr, uint32_t data)
{
    RTCState *s = opaque;
    if ((addr & 1) == 0) {
        s->io_info = OUTB_0x70; // for temporal property
        s->cmos_index = data & 0x7f;
    } else {
        switch(s->cmos_index) {
            case RTC_SECONDS_ALARM:
                #ifdef RTC_BENCHMARK_PROP_9
                    assert((s->cmos_data[RTC_REG_B] & REG_B_SET) == REG_B_SET);
                #endif
        ...
    }
```

```c
```
Example Assertion

```c
void cmos_ioport_write(void *opaque, uint32_t addr, uint32_t data) {
    RTCState *s = opaque;
    if ((addr & 1) == 0) {
        s->io_info = OUTB_0x70; // for temporal property
        s->cmos_index = data & 0x7f;
    } else {
        switch(s->cmos_index) {
            case RTC_SECONDS_ALARM:
                #ifdef RTC_BENCHMARK_PROP_9
                    assert((s->cmos_data[RTC_REG_B] & REG_B_SET) == REG_B_SET);
                #endif
                ...
```
Found bug in RTC hardware model

rtc: Only call rtc_set_cmos when Register B SET flag is disabled.

This bug occurs when the SET flag of Register B is enabled. When an RTC data register (i.e. any of the ten time/calendar CMOS bytes) is set, the data is (as expected) correctly stored in the cmos_data array. However, since the SET flag is enabled, the function rtc_set_time is not invoked. As a result, the field base rtc in RTCState remains uninitialized. This causes a problem on subsequent writes which can end up overwriting data. To see this, consider writing data to Register A after having written data to any of the RTC data registers; the following figure illustrates the call stack for the Register A write operation:

```
+-- cmos_io_port_write
   +-- check_update_timer
   +---- get_next_alarm
       +----- rtc_update_time
```

In rtc_update_time, get_guest_RTC calculates the wrong time and overwrites the previously written RTC data register values.

Signed-off-by: Alex Horn <alex.horn@cs.ox.ac.uk>
Signed-off-by: Paolo Bonzini <pbonzini@redhat.com>
Signed-off-by: Anthony Liguori <aliguori@us.ibm.com>
Also found bug in TMP105 hardware model

```
tmp105: Fix I2C protocol bug

author       Andreas Färber <andreas.faerber@web.de>
Wed, 16 Jan 2013 00:57:56 +0000 (01:57 +0100)
committer    Anthony Liguori <aliguori@us.ibm.com>
Wed, 16 Jan 2013 18:14:20 +0000 (12:14 -0600)
commit        cb5ef3fa1871522a0886627033459e94bd537fb7
              tree    ec94c5b0f0514297227eb6de0a0e432f2affe5a2
              parent  6d0b430176e3571af0e1596276078f05bf6c5a5

tmp105: Fix I2C protocol bug

An early length postincrement in the TMP105's I2C TX path led to transfers of more than one byte to place the second byte in the third byte's place within the buffer and the third byte to get discarded.

Fix this by explicitly incrementing the length after the checks but before the callback is called, which again checks the length.

Adjust the Coding Style while at it.

Signed-off-by: Alex Horn <alex.horn@cs.ox.ac.uk>
Signed-off-by: Andreas Färber <andreas.faerber@web.de>
Reviewed-by: Anthony Liguori <aliguori@us.ibm.com>
Signed-off-by: Anthony Liguori <aliguori@us.ibm.com>
```
Experiments

Hardware/software interface properties formally checked on an individual basis:

- 11 RTC properties within a few minutes
- 17 TMP105 properties in less than 15 minutes
- 3 Ethernet MAC properties in sequential code within a few minutes
- 7 Ethernet MAC properties in concurrent code within a few hours

---

After publication, we found a bug in CBMC’s implementation of the partial order concurrency encoding but continue to improve the code. At the present time, we cannot reproduce the results with CBMC for the concurrent model of the Ethernet MAC.
Download Me!

Conclusion:

- Formal verification of hardware/software interface properties written in C code
  - Executable code leverages well-established testing principles in industry
  - Apply multi-path (i.e. CBMC-style) symbolic execution and symbolic partial order encodings to handle concurrency in hardware/software

- Open-source prototype of a verifiable virtual platform (VVP)
  - Provides an object of study for software engineers

All code and documentation is openly available now.

Source code, data sheets and experiments can be downloaded at http://www.cprover.org/firmware/.
Conclusion:

- Formal verification of hardware/software interface properties written in C code
  - Executable code leverages well-established testing principles in industry
  - Apply multi-path (i.e. CBMC-style) symbolic execution and symbolic partial order encodings to handle concurrency in hardware/software
- Open-source prototype of a verifiable virtual platform (VVP)
  - Provides an object of study for software engineers

All code and documentation is openly available now.

Source code, data sheets and experiments can be downloaded at http://www.cprover.org/firmware/.

Thank you.
Symbolic partial order encoding with CBMC

(Not to be confused with partial order reduction.)

Source Code

\[ x := 1; \]

Static Single Assignment Form

\[ x_0 = 1 \]

Symbolic Event Structure

\((m1) W x x_0\)

Symbolic Partial Orders

\(\text{clock}_{m1}\)

Decision Procedure

(SAT/SMT)

CAV’13
RTC/QEMU: QDev and QOM Simplifications

Domain knowledge required:

system bus

- hpet
- fw-cfg
- i440FX-pci-host
- ioapic
- apic

pci.0

- i440FX
- PIIX3
- piix3-ide
- piix3-usb-uhci
- PIIX4_PM
- cirrus-vga

i440FX

- isa.0
- i8042
- mc146181rtc

isa-fdc

- ide.0
- ide.1
- usb.0
- i2c

smbus-eeprom etc.
Related work by Kai Cong et al.

Recent Kai Cong, Fei Xie, and Li Lei publications:

- **Symbolic Execution of Virtual Devices (QSIC, 2013).**
  - Single-path (KLEE-style) symbolic execution
  - Doesn’t symbolically co-execute virtual device and driver
  - Suggests a way to automatically extract QEMU hardware models

- **Automatic Concolic Test Generation with Virtual Prototypes for Post-silicon Validation (ICCAD, 2013).**
  - Uses QEMU and KLEE
  - Records concrete hardware/software interactions