High-Fidelity Specification of Real-World Devices

Liam Murphy
liam.p.murphy@student.unsw.edu.au
UNSW Sydney
Australia

Albert Rizaldi
albert.rizaldi@planv.tech
PlanV GmbH
Germany

Lesley Rossouw
lesley.rossouw@unsw.edu.au
UNSW Sydney
Australia

Chen George
gchen@cs.wisc.edu
University of Wisconsin - Madison
U.S.A.

James Treloar
james.treloar@student.unsw.edu.au
UNSW Sydney
Australia

Hammond Pearce
hammond.pearce@unsw.edu.au
UNSW Sydney
Australia

Miki Tanaka
miki.tanaka@unsw.edu.au
UNSW Sydney
Australia

Gernot Heiser
gernot@unsw.edu.au
UNSW Sydney
Australia

Abstract

Device driver bugs are the leading cause of operating-system exploits, and the lack of accurate specifications of device interfaces is a leading cause of driver bugs. We propose to address the specification issue by deriving formal specifications of devices from their Verilog implementation, and prove the correctness of the specification against the implementation. We demonstrate this approach by applying it to an open-source I^2C controller. These specifications should enable synthesis or verification of drivers in the future.

CCS Concepts: • **Software and its engineering** → **Operating systems**; **Formal software verification**; **Formal methods**; • **Hardware** → *Theorem proving and SAT solving*.

Keywords: Device model, device driver verification, HOL4

ACM Reference Format:
Liam Murphy, Albert Rizaldi, Lesley Rossouw, Chen George, James Treloar, Hammond Pearce, Miki Tanaka, and Gernot Heiser. 2025. High-Fidelity Specification of Real-World Devices. In *13th Workshop on Programming Languages and Operating Systems (PLOS '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3764860.3768335

1 Introduction

Bugs in device drivers are the major source of OS vulnerabilities, accounting for the majority of the 1,057 CVEs reported for Linux in the period 2018–22 [Pohjola et al. 2023] – correct drivers are therefore critical to the secure and safe operation of any computing system.

Driver bugs could be largely eliminated by synthesising drivers from formal specification of the hardware and OS interfaces [Ryzhyk et al. 2009b], or by formally verifying hand-written drivers [Pohjola et al. 2023]; such verification is our ultimate aim. However, synthesis or verification of drivers on its own is insufficient: A study by Ryzhyk et al. [2009a] showed that the dominant cause of Linux driver bugs (38%) are *device-protocol violations*, i.e. the driver interfacing with the device incorrectly, typically the result of incorrect or mis-understood specifications of the device interface.

The remaining faults distribute roughly equally between software-protocol (i.e. OS interface) violations, concurrency faults and "generic faults" (coding errors). Based on this evidence, Ryzhyk et al. [2009a] propose an *active driver* model, where each device driver is a single-threaded, event-driven process, communicating with the rest of the OS via well-defined interfaces; such a approach eliminates most of the software-protocol and concurrency bugs (which together account for 39% of bugs). While their proposed Dingo driver framework failed to get traction in the Linux community, the seL4 community has recently introduced a device driver model that follows the Dingo approach, resulting in simple drivers for seL4 that outperform Linux [Heiser et al. 2024].

While there is currently no quantitative data on the correctness benefits of the active driver model, it is highly intuitive that its adoption will reduce the incidence of the classes of bugs it targets, *resulting in device-protocol violations becoming the even more dominant source of driver bugs*. The Dingo study further states that major sources of these bugs are *poorly documented device behaviour, and the device deviating from hardware interface standards* or otherwise documented behaviour (incl. hardware bugs).

Clearly, precise and correct specification of device interfaces are a prerequisite for eliminating these device-protocol violations. We propose to not only *formally specifying* device

PLOS '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2225-7/25/10
https://doi.org/10.1145/3764860.3768335

interfaces, but also *verifying* this specification against the device implementation. Our long-term aim, outside of the scope of this paper, is to formally verify the device driver. However, even without full-scale formal verification, high-fidelity formal specification of the device will enable the construction of better drivers, be it manually or by synthesis.

The **contributions** of this work are:

1. a framework for deriving validated formal device specifications in the HOL4 theorem prover from Verilog implementations, consisting of:
   a. a general workflow for proving that the Verilog implementation of a device refines a *higher order logic* (HOL) specification (Section 3.1);
   b. a framework for defining the HOL specifications from Verilog implementations (Section 3.4);
2. application of the framework to a pre-existing, open-source I^2C controller (Section 4).

Specifically, we present the high-level device specifications as parametrised HOL functions that capture the basic structures of the devices, where the parameters are then instantiated with the details specific to each device.

We also produce a HOL representation of the Verilog implementation of a device, either from the original Verilog design, or a manually derived simplification of the Verilog.

We then present the proof of equivalence between the HOL specification and the HOL representation of Verilog. We furthermore show how to prove the behavioural equivalence (i.e., equivalence of the device interface and hardware-software interaction protocol) between this Verilog representation in HOL4 and the original Verilog design, which thus establishes that the Verilog refines the specification.

We present formalisation of a real-world I^2C controller, for which the above proof is completed for the address-decoding logic part, from the specification down to the original Verilog.

2 Tools and Device Implementations Used

2.1 Verilog Syntax and Semantics in HOL4

We use the theorem prover HOL4 [Slind and Norrish 2008] as the basic tool for formalisation and verification. The definitional framework of HOL4 is higher order logic (HOL), which is essentially a functional programming language. We use two kinds of Verilog representations on HOL4: one is *shallowly embedded* Verilog, meaning that Verilog functionality is represented semantically and directly as HOL4 functions, and the other is *deeply embedded* Verilog, which is defined as a datatype in HOL4 representing Verilog syntax trees.

Lööw and Myreen [2019] presents deeply embedded Verilog and its semantics in HOL4 [Lööw 2018]. It also provides a verified synthesis tool (down to netlists) and a proof-producing Verilog code generator. We use this definition of deeply embedded Verilog in our formalisation. We also use the Verilog code generator as a "translator" from shallowly embedded Verilog to a deeply embedded version, which we then can export as Verilog files. The proof produced by this translation process guarantees that the properties established on the HOL4 specification hold for the Verilog design obtained as the result of translation. We do not plan to use the synthesis tool because our workflow presumes the presence of an existing hardware design. We will refer to the proof-producing translator and the Verilog semantics from this work simply as the (Verilog) translator and the Verilog semantics in the rest of the paper.

2.2 Target Platform

We target the OpenTitan Foundation's secure peripherals, specifically a fork [OpenTitan Developers 2021] of their I^2C created by the PULP group for use in the Cheshire open-source RISC-V SoC [Ottaviano et al. 2023].

3 Specification Framework

The general workflow we discuss in Section 3.1 is, in principle, applicable to generic Verilog designs. For designs by the PULP project, we present some general structures that provide guidance in defining HOL specifications (Section 3.4).

3.1 General Workflow

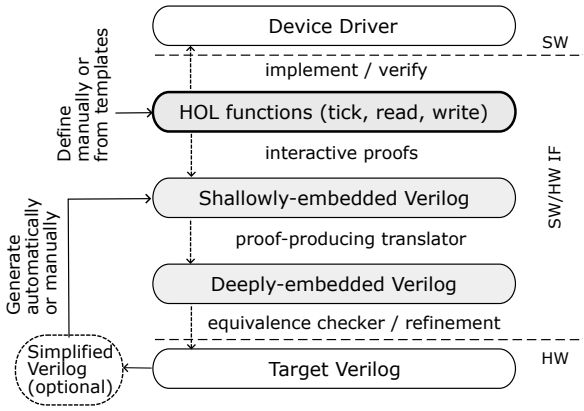

Figure 1. Device interface formalisation.

Figure 1 shows the workflow. The *HOL functions* box represents the formal specification, against which we could implement (and eventually verify) device drivers. The bottom box is the Cheshire design in SystemVerilog. The refinement between the HOL specification and the original Verilog is established using HOL representations of Verilog, which serves as an intermediate representation. We use two different HOL representations of Verilog: shallowly embedded and deeply embedded (the second and third from the bottom; see Section 2.1 for definitions).

It may be necessary to manually translate the target design into a simplified version of Verilog, which only uses features that are supported by the current Verilog semantics, e.g., no submodules (on the left).

The HOL functions ("the spec") are manually defined based on the template structures and the (Cheshire) device-specific information. We then show in three steps that the spec behaves like the original target Verilog:

1. We show, by manual proofs in HOL4, the equivalence of behaviours between the spec and the shallowly embedded version of Verilog, which is obtained by manually or automatically translating the target Verilog (or simplified Verilog).
2. We use the Verilog translator [Lööw and Myreen 2019] to generate deeply embedded Verilog. It also produces the proof of the correctness of this translation, establishing that the two versions of Verilog are equivalent.
3. We export the deeply embedded Verilog as a Verilog file, and use any of the standard formal verification approaches used for hardware (e.g., equivalence checkers such as Yosys EQY [Yosys Developers 2020]) to show its equivalence or refinement to the target Verilog.

3.2 Top-level Specification

Typical synchronous circuits consist of two parts: storage elements (flip-flops and SRAM), whose values are updated on every rising clock edge, and combinational logic (ordinary logic gates), which determines what those storage elements' values should be changed to. The values of storage elements are referred to as synchronous signals, and intermediate values within the combinational logic are referred to as combinational signals.

Such circuits can naturally be modelled with a "state" datatype `s`, representing all the synchronous signals, and a function $\mathtt{f} : \mathtt{s} \rightarrow \mathtt{s}$ on states that models the changes made to those signals by the combinational logic each clock cycle.

However, this as is does not take into account the effects of external inputs on the circuit or expose the outputs from the circuit. Therefore, for inputs which are controlled by the device driver (i.e. memory-mapped I/O requests), we add an additional argument to `f` representing the values of these inputs. We model the circuit's remaining inputs as nondeterminism by adding an extra field to the state that carries an infinite stream of numbers `fnums`, which `f` can use to emulate non-deterministic values. This encodes a set of possible new states, based on the set of all results produced by different values of `fnums`. We represent outputs by extending the return value of `f` with an additional value. Combinational signals are represented as intermediate values private to `f`.

We realise the resulting function $\mathtt{f} : \mathtt{s} \rightarrow \mathtt{i} \rightarrow (\mathtt{s}, \mathtt{o})$ as the top-level specification `cheshire_run`, which is parametrised by three functions `tick`, `read`, and `write` that specify the device-specific behaviour:

```
cheshire_run tick read write:
     state -> req option list ->
           ffi_outcome + state # word32 option list
```

This accepts the initial state of the circuit and a list of requests to issue on each clock cycle (or `NONE` if no request should be performed), and returns one possible state of the circuit after executing it with those requests (note that `cheshire_run` executes as many cycles as the length of the requests list and returns an output list of the same length).

3.3 Cheshire Peripherals

Cheshire peripherals are split into two components: their address-decoding logic/register storage (the `*_reg_top` submodule), and their core logic (the `*_core` submodule). The address-decoding logic is auto-generated from a HJSON file that lists the peripheral's memory-mapped I/O registers along with the metadata of whether software and/or hardware is allowed to read from/write to each of the registers. The core logic communicates with the bus entirely through the interface provided by the address-decoding logic.

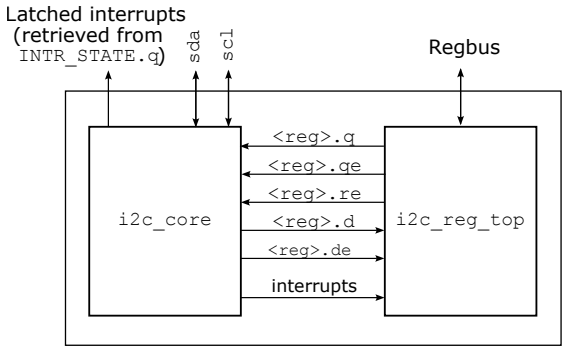

Figure 2. Cheshire I^{2}C architecture.

As an example, Figure 2 shows this structure for the I^{2}C device, but this overall structure is shared among different Cheshire devices: in `*_core`, the `d` and `de` signals are used by the core logic to set registers, and the `q` signal exposes their current values. In addition, the `qe` and `re` signals are asserted whenever software writes to or reads from a register respectively, allowing for actions to be reliably triggered by register reads or writes.

There are two kinds of registers: normal ones, which are stored by `*_reg_top`, and `hwext` ones, which are stored by `*_core`. Everything to do with `hwext` registers happens one cycle sooner than for normal registers, since `*_core` is directly responsible for servicing reads/writes rather than being notified after the fact.

3.4 Specification Parameters

.

The `tick`, `read` and `write` functions define the behaviour of a peripheral. The `tick` function defines the behaviour of the core logic. This accepts a user-defined `state` record and a *notification* representing `qe`/`re` for `hwext` registers, and returns the new state of the hardware after one clock cycle.

```
periph_state = <|
  fnums: num -> num;
  regs: periph_regs;
  fifo: word8 list;
  buffered_notif: periph_notif option;
  count: word32;
|>;

periph_tick (hwext_notif: periph_hwext_notif option)
            (st: periph_state) =
let
  (* q (non-hwext) *)
  count' = st.count + st.regs.step.step;
  (* qe (non-hwext) *)
  count'' = if st.buffered_notif = SOME step_write
            then 0w else count';
  (* d/de (non-hwext) *)
  regs' := st.regs with
      my_reg := st.regs.my_reg with my_field := fnums 0;

  fifo' = case hwext_notif of
       (* q / qe (hwext) *)
       SOME (Write (fifo_write value))
                    => st.fifo ++ [value.fifo]
       (* re (hwext) *)
     | SOME (Read fifo_read) => TL st.fifo
     | NONE => st.fifo
  fnums' := \n. st.fnums (n + 1);
in
  <|
    fnums := fnums';
    regs := regs';
    fifo := fifo';
    buffered_notif := NONE;
    count := count'';
  |>

(* d (hwext) *)
periph_get_fifo_fifo (st: periph_state) = HD st.fifo
```

Figure 3. A sample model of a peripheral's core logic.

For non-`hwext` registers, because `qe`/`re` are asserted a cycle after the read/write occurs, the notifications are saved into `buffered_notif` to delay them for a cycle, too. Figure 3 provides an example of these mechanisms in use.

We also need a getter function for each readable `hwext` field, which determines the value to be read by software, corresponding to `d`. We then generate from the peripheral's memory map a `read` and a `write` function, the model of the address-decoding logic. Figure 4 and Figure 5 show snippets of these functions generated for the I^{2}C device.

`read` takes the state of the system, the number of bytes and the address to read, and returns an optional notification that a `hwext` register has been read (corresponding to `re`) along with the read value. For `hwext` registers, it calls the getter functions (in the core logic) to determine their values.

`write` takes the same arguments as `read` plus the data to be written, and returns `st_upd`, a function to be applied after the rest of the clock cycle's logic has run, and another

```
i2c_read (st: i2c_state) (nb: num) (offset: num) =
case offset of
  0x0 =>
  INR (NONE, (w2w st.regs.intr_state.fmt_threshold
              <<~ 0w) || (* ... *): word32)
| 0x4 =>
  INR (NONE, (w2w st.regs.intr_enable.fmt_threshold
              <<~ 0w) || (* ... *): word32)
| 0x8 => INR (NONE, 0w: word32)
(* ... *)
| 0x14 => INR (NONE, (w2w (i2c_get_status_fmtfull st)
              <<~ 0w) || (* ... *): word32)
| 0x18 => INR (SOME (Read rdata_read),
              (w2w (i2c_get_rdata_rdata st)
                <<~ 0w): word32)
(* ... *)
| _ => INL FFI_failed
```

Figure 4. Snippets of the generated `i2c_read` function.

optional notification corresponding to `qe`. However, this notification is only returned for `hwext` registers: for regular registers, it needs to be delayed by a clock cycle, which is accomplished by saving the notification in `buffered_notif`.

`st_upd` is necessary to avoid a race condition: in hardware, the address-decoding and core logic are running in parallel, and are operating on the values of `regs` and `buffered_notif` set by the previous clock cycle. But if `tick` and `write` updated the state they read from, the only way to apply both their updates to a state would be to give one function the result of the other, causing it to observe changes that should not have been applied until the next clock cycle. `st_upd` decouples the state that changes are applied to from the one which determines the changes to apply.[1]

Some of the patterns of this framework are robust enough to auto-generate the HOL spec, particularly for the `*_reg_top` part. For the `*_core` part, the user needs to provide the device logic in the `*_tick` function. Figure 3 shows the basic pattern for this, outlining how different structures correspond to interacting with `*_reg_top`'s interface.

The user also needs to add whatever of the peripheral's abstract state required to model to the `state` record. They can then fill in the getter functions `*_get_*` with appropriate logic as needed for determining those registers' values.

4 Formal Specification of I^{2}C

We now apply the framework to OpenTitan's I^{2}C device used by Cheshire, and discuss the process of showing equivalence between the representations of Figure 1.

4.1 The Device

An I^{2}C device [NXP 2021] is a synchronous serial bus widely used in most contemporary computers and embedded systems, especially in common derivatives like SMBus. It uses

[1]Another option would have been to add an extra `st'` parameter to `write`, but that would not allow the notification to be passed to `tick` without having to know the result of `tick` first.

```
i2c_write (st:i2c_state) (nb:num) (off:num) (w:word32) =
case off of
  0x0 =>
    let
      st_upd = \st'. st' with <|
        regs := st'.regs with
          intr_state := st'.regs.intr_state with <|
            fmt_threshold :=
                    st'.regs.intr_state.fmt_threshold
                    && ~((0 >< 0) w);
            rx_threshold :=
                    st'.regs.intr_state.rx_threshold
                    && ~((1 >< 1) w);
            (* ... *) |>;|>;
    in
      if nb >= 2 then INR (st_upd, NONE)
      else INL FFI_failed
(* ... *)
| 0x1c =>
    let
      st_upd = \st'. st' with <|
      regs := st'.regs with fdata := st'.regs.fdata with
        <| fbyte := (7 >< 0) w;
           start := (8 >< 8) w;
           (* ... *) |>;
        buffered_notif := SOME fdata_write; |>;
    in
      if nb >= 2 then INR (st_upd, NONE)
      else INL FFI_failed
(* ... *)
| _ => INL FFI_failed
```

Figure 5. Snippets of the generated `i2c_write` function.

addressing: in our configuration, there are 7 address bits targeting up to 127 devices, with address 0 indicating broadcast. Usually there is one *controller* and many *target* devices, each with a unique address. We use I^{2}C in the *controller* mode as appropriate for the CPU.

I^{2}C uses two signal wires: SDA (serial data) and SCL (serial clock), see Figure 2. One bit is transmitted via SDA for every SCL cycle. The controller generates the clock and emits the addresses; targets are passive until addressed and clocked.

4.2 Spec and Shallow Embedding

In this section, we explain the process we used for generating the I^{2}C device specification, as HOL functions, and the shallow embedding of the Verilog design, for each of the two components of I^{2}C, namely `i2c_reg_top` and `i2c_core`.

For the `i2c_reg_top` submodule, which is the address-decoding logic part, we use Python scripts to read the HJSON file that describes the device's memory map and auto-generate both the spec and the shallow embedding, along with some necessary information. Specifically, the scripts generate the following:

- A record type for storing the non-`hwext` registers.
- `i2c_read` and `i2c_write`, which take in memory-mapped I/O requests and return:
  - A "notification" about the request which should be passed to `i2c_tick`, if needed (including requests for it to service `hwext` writes)
  - If this is a write, modifications that should be made to the state as a result; and
  - If this is a read, the value that was read (`hwext` reads are implemented by calling into user-written getter functions).
- A shallowly embedded Verilog version of `i2c_reg_top`, as well as the datatypes it uses to communicate with the shallowly embedded version of `i2c_core`.

Note that the last item is for shallow embedding, while the rest are part of the spec. These are put together with the corresponding parts for `i2c_core` to represent the top-level functionality of the device as the spec and the shallow embedding of the device design, respectively.

For `i2c_core`, the process is more manual. To model `i2c_core` as `i2c_tick` function (Figure 3), we study the simplified version of Verilog design and rewrite it as HOL4 expressions, leveraging HOL mechanisms such as datatype definitions and pattern matching. This means that FIFOs are represented as inductive lists with auxiliary functions to determine their statuses (empty, full, or in between), taking care to ensure that any function applied to the lists does not break the length invariant. External world is left underspecified via the non-deterministic function `fnums`.

For example, the SCL and SDA signals are written as `scl_i = n2w $ fnums 0` and `sda_i = n2w $ fnums 1`, where `n2w` is the HOL4 function to convert naturals to machine words. Internal registers appear in the model as fields of the record `i2c_state` whose types are machine words with suitable length.

We similarly obtain the shallowly embedded version of Verilog for `i2c_core` by defining functions in HOL4 but with more concrete representations and as a more direct mapping of Verilog designs. Unlike the specification, shallowly embedded Verilog is split into many functions describing small pieces of the circuit, corresponding to Verilog processes. Each process is either combinational or synchronous, and sets the corresponding kind of signal (as described in Section 3.2). With the combinational logic spread across multiple functions, its internal signals can no longer be kept private to one function; therefore, combinational signals must be included in the state. The circuit's outputs are also included in the state, and may be either combinational or synchronous.

On each clock cycle, all of the synchronous processes are run in one phase, followed by all of the combinational processes in another phase. Each of these processes is represented as a HOL function with three record-type arguments, which represent the current values of input signals (input `i` to the state-transition function `f`), the circuit's state at the

start of this phase (s) and the not-yet-complete value of the circuit's state at the end of each phase (s, o).

The body of each function describes the behaviour of the intended circuit by updating the fields of the third record accordingly, using HOL operations that more or less correspond to those of (System)Verilog. Together with an expression of the initial value of each signals, they constitute the top-level of a shallowly embedded Verilog circuit in HOL4.

Unlike the spec, where we can simply use lists to represents FIFOs, we need to implement FIFOs concretely for the shallow embedding. This amounts to recreating the FIFOs faithfully to the OpenTitan designs as described above. Shallowly embedded Verilog circuits in HOL4 also do not have the luxury of datatype mechanisms, meaning that the states in finite state machines need to be encoded explicitly, i.e., assigning `0b01` for `Active`, `0b11` for `ReadClockPulse`, etc.

4.3 Translation of Shallow Embedding

For each of `i2c_reg_top` and `i2c_core`, once the setup in Section 4.2 for the shallow embedding is done, we can use the proof-producing Verilog translator [Lööw and Myreen 2019] to generate the deeply embedded Verilog together with the proof that the `i2c_reg_top` + `i2c_core` circuits in both embeddings behave equivalently. Once the deeply embedded Verilog is available, we can use the pretty-printer [Lööw and Myreen 2019] to generate readable Verilog code and export it for further formal verification treatment with hardware model checkers. The translation took 63 minutes on a machine with an 8-Core AMD processor (2 threads per core) with 64GB of RAM.

4.4 Proof of Equivalence

For the `i2c_reg_top` submodule, we have completed all the three steps of Section 3.1, resulting in a proof of behavioural equivalence between the spec and the original Verilog design. Here, we discuss the first step, the manual proofs in HOL4 between the spec and the shallow embedding.

Because `i2c_reg_top`'s job is to interact with the outside world, verifying it just amounts to proving I^{2}C's top-level correctness theorem, with the correctness of `i2c_core` as an assumption. The statement of this top-level theorem without the said assumption is shown in Figure 6.

It says that, given an initial state `st` and a sequence of optional memory-mapped I/O requests `reqs` (one per clock cycle), if the initial states of the model and shallowly embedded Verilog are equivalent, and all the requests made to the model and the Verilog are equivalent and valid, then there exists some `fnums` that can be passed to the model such that after running it and the Verilog with `reqs`, their states and all the values that they return to software will be equivalent.

This amounts to say that, if the shallowly embedded Verilog version of `i2c_core` behaves the same as `i2c_tick`, the `i2c_reg_top` model behaves the same as the whole of `i2c` (which is a thin wrapper around `i2c_core` + `i2c_reg_top`).

```
i2c_state_rel st (i2c fext fbits 0)
                 /\ (!i. i < LENGTH reqs ==>
  let
    req = reg_req_decode (fext i).reg_req_i: 7 reg_req;
  in
    ~i2c_req_error req
     /\ cheshire_req_rel (EL i reqs) req) ==>
 ?fnums. let
    (st', rdatas) =
    OUTR (cheshire_run i2c_tick i2c_read i2c_write
                       (st with fnums := fnums) reqs)
  in
  i2c_state_rel st' (i2c fext fbits (LENGTH reqs))
  /\ (!i. i < LENGTH reqs ==>
    let
      rsp = reg_rsp_decode (i2c fext fbits i).reg_rsp_o;
    in
      rsp.ready /\ ~rsp.error /\
      (!value. EL i rdatas = SOME value
                       ==> rsp.rdata = value))
```

Figure 6. The correctness theorem for Cheshire I^{2}C's shallowly embedded Verilog.

Because the address-decoding logic is auto-generated in the same fashion for all of Cheshire's peripherals, we should be able to verify it for one peripheral and reuse that proof for the rest of them with minimal modifications.

5 Discussion

5.1 Status

We now have a complete formal specification and shallow embedding of the Cheshire I^{2}C controller. For the `i2c_reg_top` submodule, we have a complete proof of behavioural equivalence down to the original Verilog (RTL level design); Figure 6 shows the theorem. The proof for the `i2c_core` part is in progress. We are also working on a HOL4 tool to help establishing refinement between simplified Verilog and the original Verilog.

We are also working on repeating the process for a second device class, Cheshire's SPI controller. Generation of the spec and shallow embedding progressed as for I^{2}C, except that the Python scripts to handle the HJSON file needed to be extended to support window registers. This provides some confidence that the approach generalises over a large set of device classes.

We emphasise that our targets are not toy examples but real-world devices of reasonable complexity. For reference, we list the numbers of lines of the Verilog designs that we use, the device drivers written in C, and the HOL specifications for both I^{2}C and SPI devices in Table 1. Just to put things in perspective, we also add numbers for an Ethernet controller. (Note, however, the design and the driver are unrelated in the case of the Ethernet controller; the Verilog code is from the PULP project [PULP Developers 2023], while the driver is for the Amlogic Meson SoC's Ethernet peripheral.)

	Verilog	Driver	HOL spec
I²C	5993	713	1414
SPI	4609	864	1235
Ethernet	3987	641	N/A

Table 1. Lines of code comparisons between I²C, SPI, and Ethernet devices

5.2 Future work

We expect the specification structures to cover any peripherals used by the PULP project. Our next aim is to formalise an open-source Ethernet controller from a different source, which will likely require some extension of the framework.

For usability and productivity we hope to reduce the manual process steps required. The Verilog translator works in the opposite direction of what is ideal for our purpose: we would prefer an automated tool that generates shallow embedding from deep embedding, i.e. a *decompiler*. Also, extending the Verilog semantics to support submodules will reduce the need to prepare a simplified version of Verilog.

Irrespective, the availability of formal, verified hardware specifications opens the exciting prospect of not only *reducing* faults in device drivers, but completely *eliminating* them by formally verifying drivers against the spec. This long-term goal is aided by the structural simplification of sDDF drivers [Heiser et al. 2024] (compared to Linux drivers).

We hope to implement device drivers based on the formal specification we report here. At the time of writing, we already have an I²C device driver implemented in C, based on the OpenTitan Verilog implementation. Our immediate plan is to assess this implementation against the formalisation we present here to see if we find any discrepancies or possible improvements, potentially in both ways. We aim to do the same for the SPI driver as well.

Our formalisation also paves the way to formally verifying properties of the hardware peripheral, not only in isolation, but also in conjunction with its driver. These could include safety properties (e.g. bus-level correctness and bit-framing including clock stretching), liveness properties (including worst case response times and deadlock checks), and interface properties (including register memory mapping, memory read/write protection, and interrupt sequencing).

6 Related work

Attempts to formalise and verify hardware interfaces go back to Bevier et al. [1989] who verified a simple processor against its ISA, with an assembler and compiler; I/O was not included in their formalisation. The Verisoft project verified a processor and a simple microkernel [Alkassar et al. 2008]. Verisoft verified some drivers, although against abstract models not connected to the hardware implementation.

Termite synthesised device drivers from (manually derived) formal specifications of the OS and hardware interfaces [Ryzhyk et al. 2009b, 2014]. While scaling to the complexity of Ethernet drivers, Termite could not handle direct memory access (DMA) by the device.

Lööw et al. [2019] pioneered verifying hardware specs against the Verilog description of hardware. While they verified execution of (bare-metal) code, they did not model I/O.

Erbsen et al. [2021] verified a (bare-metal) stack with realistic I/O down to the hardware for the first time. However, they did not perform any modelling or verification of the peripherals connected to the memory-mapped I/O bus.

Athalye et al. [2022] created a hardware/software verification framework aimed at proving the absence of timing channels, which verifies in one go with an SMT-solver-based tool that the entire system, firmware included, functions correctly. This means that any change to the software requires the whole system to be verified again. In follow-up work [Athalye et al. 2024] they improve the scalability by verifying the software separately.

All of the above work has in common that it is based on simplified hardware and software. In most cases, all software must be trusted to gain an overall verification story. Our aim is to produce dependable (and performant) device drivers for verified real-world devices for a system running untrusted code (isolated by the OS).

7 Conclusions

We presented a framework for formally specifying peripheral devices as HOL functions and demonstrated its use in formalising an open-source I²C controller from its Verilog implementation and verifying the spec against the Verilog. Using the same workflow we have generated a specification for an SPI controller, with verification partially completed, indicating that the approach works across device classes.

Such formal specifications address the leading cause of device driver bugs, device protocol violations, and should thus lead to better (manually-written or synthesised) drivers. More importantly, they are a core requirement for meaningful formal verification of the device drivers – the only way to completely remove driver bugs.

Availability

The tools and artefacts from this work are available under open-source licenses from https://trustworthy.systems/software/device-spec/.

Acknowledgments

This work was performed under the PISTIs-V project, funded by the German agency *Agentur für Innovation in der Cybersicherheit GmbH* (Cyberagentur) under the *Ecosystem Formally Verifiable IT – Provable Cybersecurity* (EVIT) Program.

References

Eyad Alkassar, Mark Hillebrand, Dirk Leinenbach, Norbert Schirmer, and Artem Starostin. 2008. The Verisoft Approach to Systems Verification. In *Verified Software: Theories, Tools and Experiments (Lecture Notes in Computer Science)*, Vol. 5295. Springer, 209–224.

Anish Athalye, Henry Corrigan-Gibbs, Frans Kaashoek, Joseph Tassarotti, and Nickolai Zeldovich. 2024. Modular Verification of Secure and Leakage-Free Systems: From Application Specification to Circuit-Level Implementation. In *ACM Symposium on Operating Systems Principles*. New York, NY, USA, 655–672.

Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with Information-Preserving Refinement. In *USENIX Symposium on Operating Systems Design and Implementation*. 503–519. https://www.usenix.org/conference/osdi22/presentation/athalye

William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. 1989. An approach to systems verification. *Journal of Automated Reasoning* 5, 4 (1989), 411–428.

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Gernot Heiser, Peter Chubb, Alex Brown, Julia Broady, Courtney Darville, and Lucy Parker. 2024. The seL4 Device Driver Framework (sDDF). https://trustworthy.systems/projects/drivers/sddf-design.pdf

Andreas Lööw. 2018. Verilog development and verification project for HOL4. https://github.com/CakeML/hardware

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Phoenix, AZ, US, 1041–1053.

Andreas Lööw and Magnus O. Myreen. 2019. A proof-producing translator for Verilog development in HOL. In *Proceedings of the International Workshop on Formal Methods in Software Engineering (FormaliSE@ICSE)*. 99–108.

NXP 2021. UM10204: I^2C-bus specification and user manual, Rev. 7.0. https://www.nxp.com/docs/en/user-guide/UM10204.pdf

OpenTitan Developers. 2021. Selected peripherals from OpenTitan with PULP patches. https://github.com/pulp-platform/opentitan_peripherals

Alessandro Ottaviano, Thomas Benz, Paul Scheffler, and Luca Benini. 2023. Cheshire: A Lightweight, Linux-Capable RISC-V Host Platform for Domain-Specific Accelerator Plug-In. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 10 (2023), 3777–3781.

Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O. Myreen, Michael Norrish, and Gernot Heiser. 2023. Pancake: Verified Systems Programming Made Sweeter. In *Workshop on Programming Languages and Operating Systems (PLOS)*. Koblenz, DE.

PULP Developers. 2023. PULP Ethernet. https://github.com/pulp-platform/pulp-ethernet

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. 2009a. Dingo: Taming Device Drivers. In *EuroSys Conference*. Nuremberg, DE, 275–288.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009b. Automatic Device Driver Synthesis with Termite. In *ACM Symposium on Operating Systems Principles*. Big Sky, MT, US, 73–86.

Leonid Ryzhyk, Adam Christopher Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-Guided Device Driver Synthesis. In *USENIX Symposium on Operating Systems Design and Implementation*. Broomfield, CO, USA, 661–676.

Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, Montréal, Canada, 28–32.

Yosys Developers. 2020. Equivalence checking with Yosys. https://github.com/YosysHQ/eqy