A model-checked I^2C specification

Lukas Humbel, Daniel Schwyn, Nora Hossle, Roni Haecki, Melissa Liccardello, Jan Schaer, David Cock, Michael Giardino, and Timothy Roscoe

ETH Zurich
`firstname.lastname@inf.ethz.ch`

Abstract. I^2C is a pervasive bus protocol used for querying sensors and actuators, but it is plagued with incompatible devices, violating the specification at various levels.
Interacting with partially compliant devices poses several challenges. Compatibility of the controller interface, as well as the driver code, must be checked manually and potentially changed. This is a difficult process, as interactions with other bus devices must also be considered. We propose a model checking approach to quickly write high-assurance drivers and layers of the I^2C stack. We do not propose a *single*, *true* formalization of I^2C, but a framework that allows to rapidly model non-compliant devices and verify the correct interaction with a host driver process.
Our contribution is twofold: First, we develop a framework that allows the specification of device and driver behavior together, and verification of their correct interaction. Second, we provide already verified, fine-grained building blocks, representing layers of the I^2C stack that can be reused to interact with partially-compliant devices, as well as reducing model checking complexity.
Our specifications are stated in a machine-readable, executable, and layered DSL. From the DSL, we generate both Promela and C code. The Promela is used to apply model checking to ensure the layer implementations follow the abstract specifications. The C code is used to build and verify an EEPROM model and driver running on a Raspberry Pi.

Keywords: model checking, serial protocol, I2C, DSL, layering

1 Introduction

We present a layered framework[1] for verifying implementations of the ubiquitous I^2C protocol and provide initial layers of the I^2C stack. Each layer has an executable implementation, formal specification, and the adherence of the implementation to the specification is model checked.

I^2C is a low-speed bus that is a fundamental building block of almost all modern computer systems. It is used to network most integrated circuits and other devices in platforms ranging from mobile phone Systems-on-a-chip (SoCs) to server motherboards. It is also used as a sideband protocol in HDMI connections and memory

[1] Source code available `http://github.com/lluki/filz`

DIMMs. While typically invisible to a machine's system software, I^{2}C is used by embedded *Baseboard Management Controllers* (BMCs) to control power and clock distribution to the rest of the computer system.

For this reason, I^{2}C is a critical (if often overlooked [1]) component. Incorrect programming of the I^{2}C network (e.g. misconfiguring a voltage regulator) can cause irrevocable hardware damage. Moreover, I^{2}C *controllers* (devices like the BMC, which initiate transactions on the network) have almost unrestricted visibility and authority over the hardware. To build a secure machine, board firmware (such as on the BMC) must be trusted. For systems as complex as modern computing platforms, real trust requires formal verification of the software stack. That, in turn, must be carried out in relation to a faithful model of the underlying hardware.

Unfortunately, I^{2}C is described in an ambiguous informal English-language document [13]. While almost all significant hardware components in a modern system talk I^{2}C, many interpret this standard differently, or only partially implement it.

Our I^{2}C specification is a first step in addressing this problem. Each logical layer in the I^{2}C protocol has a corresponding layer in the specification. At each layer, an abstract specification given as a single Promela process captures the correct behavior of the complete network (senders and receivers) at that layer. The lowest layer models electrical states on the bus and relies only on minimal timing assumptions.

In addition, deterministic, executable implementation specifications at each layer, written in a Domain Specific Language (DSL), describe end-point state machines, which are compiled into Promela. SPIN [9] is then used to verify that the abstract specification at each layer is correctly implemented by the composition of the implementations at all underlying layers. We generate C code from the executable specification which implements a complete, real-world I^{2}C stack. In Section 5.6 we show how to use this code to build a driver for an I^{2}C EEPROM.

Our specification can therefore serve as the basis for several applications and directions. Hardware designers can employ it as a rigorous, machine-checkable description of how compliant I^{2}C devices must behave, and generate high-coverage test suites for their designs. Firmware engineers can use it to generate functional, performant C code for parts of their stack, and build robust I^{2}C software implementations which can handle non-compliant devices in a robust and well-defined manner. Finally, for proof engineers seeking to do full-stack software verification of computer systems, we provide an abstract hardware model that captures the complexity of I^{2}C hardware on which to (partially) base refinement proofs of system software.

2 Background

I^{2}C is a de-facto standard [13] low-speed control bus used for connecting integrated circuits on a PCB and macrocells on an SoC. Board designers appreciate its efficiency since it uses only two shared wires, and allows much of the control sequencing of a computer system to be implemented in software by a BMC. In addition, the same bus can be used to both query sensors and control actuators, allowing for complex controller to be implemented efficiently. In this section, we describe the basic I^{2}C protocol stack and its implementation subtleties, which have motivated our specification.

An I^{2}C bus has two wires, clock (SCL) and data (SDA), which are pulled up to the supply voltage. ICs may only drive the lines low, not high. I^{2}C devices are either *controllers* or *responders*[2], and a bus can have multiple controllers and responders. Other devices (e.g. *multiplexers*), can connect different bus segments. Each responder has a bus-wide unique seven-bit address; some addresses are special and reserved. Communication is always initiated by a controller using the target responder's address.

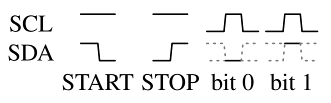

Fig. 1: The four I^{2}C bus symbols

The lowest level of the protocol uses the SCL and SDA wires to encode bits (0 or 1) and the start and end of a bus transaction as shown in Figure 1. Outside of an I^{2}C transaction, SCL is always high.

START/STOP conditions and the clock signal are generated by a controller. Responders signal a 0 bit by driving SDA low, a 1 bit by doing nothing. When a responder cannot provide the required data in a timely manner, it can perform *clock stretching* by driving SCL low during the clock low period, blocking the bus.

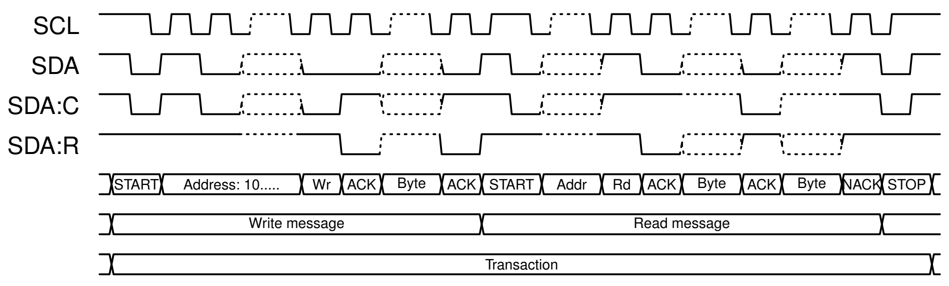

Fig. 2: Example I^{2}C transaction with two messages: write one byte, then read two bytes. SDA:C and SDA:R are SDA signals asserted by controller and responder respectively. The address and byte transfers have been abbreviated.

Above the bit layer, I^{2}C deals in *transactions* containing one or more *messages*, as shown in Figure 2. Each message begins with a START condition, and the transaction ends with a STOP condition. After each START, the controller transmits the 7-bit responder address and a bit indicating READ or WRITE. If a responder with that address is present it acknowledges with a 0 bit (ACK), otherwise the controller sees a 1

[2] In this paper, we will use the current, more precise terms 'controller' for 'master' and 'responder' for 'slave'

bit (NACK). A message will not continue after a NACK. After an ACK, the message payload follows.

If the controller sends a READ, the responder will respond to the controller with a sequence of bytes. Each byte is ACKed by the controller, otherwise the responder stops sending. Likewise, when the controller sends a WRITE, it is followed by zero or more bytes, each of which is ACKed by the responder.

The bus is only idle when both SDA and SCL are high. Collisions (two controllers starting to use the bus at the same time) are detected by a device seeing a 0 bit on the bus when it intended to transmit a 1 bit. In this case, the controllers stop transmitting and may retry the transaction later. An *undefined condition* [13, p. 12] occurs when START and STOP, START and a bit, or STOP and a bit are generated by different controllers simultaneously.

This is the complete, basic I^{2}C protocol, and appears fairly straightforward. However, many devices deviate from this standard, making it hard to capture their behavior formally. For example, the hardware I^{2}C controller in the *BCM2835* SoC (used in the Raspberry Pi 1) ignores *clock stretching* from responder devices [1], and cannot interoperate with devices that do so. The workaround is to ignore the hardware and implement the controller directly in software (known as "bit-banging"), a CPU-intensive technique. The *AS5011 Hall Sensor* [3] and the *CAT5259 Digital Potentiometer* [14] both ignore the READ/WRITE bit of a message and require every transaction to be a WRITE, while the *KS0127 video decoder* [2] ignores the STOP condition unless the controller can include it in a nonstandard position, and continues writing data on the bus. These examples are all violations of the informal protocol specification, but they occur at different layers (the bit, byte, and transaction layers).

An advantage of structuring a formal specification in layers is that conformance can be expressed up to a given layer, and then modified to accommodate the non-compliant device as a special-case above this layer.

3 Related Work

I^{2}C has served as a case study for many verification techniques, for example in applying the Analytical Software Design methodology [11]. Using a model of an existing controller device, the authors verify correct interaction between a driver and this model. Similar, Bošnački *et al.* [6] study the concurrent interactions of a Linux I^{2}C bus driver with hardware and syscalls. While our work overlaps in basic I^{2}C properties, like adherence to the addressing mode, we specify and construct the controller itself. Finkbeiner *et al.* [7] study the information flow in an existing unverified I^{2}C controller using HyperLTL, a logic that can reason about and quantify over the set of traces, and thus can correlate inputs and outputs. It is complementary to our work, since our specification could be verified against their information flow properties. In a simulation assisted verification approach [8] I^{2}C is considered. The assumptions in this work differ from ours: The system under verification is treated as a black-box and simulation is used to reduce the state-space while our goal is to replace a black box with a specified, clear system. Bos *et al.* [5] proposes to express the I^{2}C controller and devices in discrete time process algebra. Their work neither automatically verifies nor generates code. While they mention the ability to analyze deadlocks, they do not provide any conditions a higher-layer device must fulfill in order to guarantee deadlock freedom. In contrast our work's main focus is stating sufficient correctness properties for higher abstraction levels. *ACCESS.bus* is a standard that builds hotplug on top of I^{2}C. Its handshake protocol has been model checked [4]. The I^{2}C abstraction used is on a higher level (messages) than our model (down to level changes on the bus). Our work is complementary and could be used verify their assumptions. Other bus protocols that have been studied using model checking include the *CAN* bus [15] and the *AMBA* on chip bus [16]. These bus specification ensure more guarantees than I^{2}C such as fault confinement, liveness, priority-based fairness.

4 Approach and Tools

We will illustrate our approach with an example with three layers: *electrical*, *bus*, and *nibble* layer. The **BusController** receives a 4-bit *nibble* from the **NibbleController** and writes it bitwise on the electrical layer. The **BusResponder** receives these bits from the electrical layer and returns the nibble to the **NibbleResponder**. The responder either ACKs or NACKs the message.

An example exchange is given below, between *bus* and *nibble* layers. We denote x receiving value y on the lower layer with $x{\uparrow}y$, while $x{\downarrow}y$ is x sending value y on the lower layer. c is the **NibbleController** and r is the **NibbleResponder**.

$$\ldots, c{\uparrow}\text{ACK}, c{\downarrow}3, r{\uparrow}3, r{\downarrow}\text{NACK}, c{\uparrow}\text{NACK}, \ldots$$

We see c receiving an ACK (presumably from the address phase), sending a payload nibble 3, which is received and NACKed by r, ending with the NACK arriving at c. The correctness statement for this layer is that any datum written by **NibbleController** will be received by **NibbleResponder**. We do so by ensuring that the message sequence produced by the implementation and an abstract process are equal.

We implement **BusController** (Listing 1.1) and **BusResponder** (Listing 1.2) in our DSL, whose semantics are based on coroutines. Coroutines can **call** other coroutines, and the callee executes until **yield**ing to the caller. The coroutine resumes at the last **yield** when called, with the local state preserved.

We implement the bus logic in process **El**. This calls **BusController** and **BusResponder** to get their current outputs, then combines these to compute the bus state (i.e. wired-AND — the bus is 0 if *any* agent drives it low).

We verify the correctness of **BusController** and **BusResponder** (see Listing 1.3) against **BusSpec**, a nondeterministic process capturing permissible bus behavior, and **NibbleValid**, which captures allowable event sequences from the next-highest level (i.e. what the bus layer may assume). Figure 3 depicts this.

BusSpec prescribes how actions *from* the nibble layer translate into events *to* the nibble layer e.g. the number 3 in the above example. **NibbleValid** includes all possible actions at the Nibble layer. It non-deterministically transmits a 4-bit nibble, which is either ACKed or NACKed (again non-deterministically). Correct delivery is guaranteed by **BusSpec**.

```
proc (int) BusController(int res) {
    int data; int data_pos; int nibble_res;
    nibble_res = RES_ACK;
start:
    data = NibbleController(nibble_res);
    data_pos = 0;
    while(data_pos < 4){
        yield ((data >> (3-data_pos)) & 1);  //MSB first
        data_pos = data_pos + 1;      }
    yield (1);   // this reads back the ACK bit
    if(res == 0) {
        nibble_res = RES_ACK; goto start;
    } else {
        nibble_res = RES_NACK; goto start;     }
}
```

Listing 1.1: Bus Controller

```
proc (int) BusResponder(int res) {
    int buf; int read; int ack;
start:
    buf = 0; read = 0;
    while(read < 4){
        yield (1);
        assert(res == 0 or res == 1);
        buf = (buf << 1) | res;
        read = read + 1;     }
    (ack) = NibbleResponder(buf);
    yield (ack); goto start;
}
```

Listing 1.2: Bus Responder

The verifier is an additional process that polls the message channels and forwards messages to both **BusSpec** and **BusImpl**. If both produce the same result, execution continues, otherwise the verifier stalls (no transition/deadlock). Implementation correctness is then checked by using SPIN to verify the absence of deadlock in the combined process.

4.1 Programming model, DSL, and backends

As discussed, our DSL is based on coroutines. The language is (semantically) an executable subset of Promela with messages restricted to the **call** and **yield** primitives, and an acyclic call graph. These restrictions also allow for the generation of compact C code. Unlike existing C-to-Promela converters [10], we describe stateful processes (coroutines). Implicit state makes it convenient to express stateful protocols such as I^{2}C. Pro-

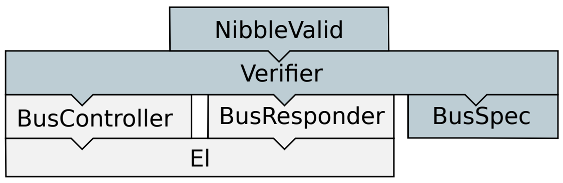

Fig. 3: Verification processes for verifying the Bus level. Gray processes are generated from the DSL; blue ones are expressed in Promela

```
proctype NibbleValid(chan ci, co, ri, ro) {
    int c_res = RES_ACK; int dat;
start:
    select(dat : 0..15);
    ci?_; co!dat;
    ri?_;
    if
    :: ro!ACT_ACK; c_res = RES_ACK; goto start;
    :: ro!ACT_NACK; c_res = RES_NACK; goto start;
    fi }

proctype BusControllerSpec(chan ci, co, ri, ro){
    int dat; int res = RES_ACK;
start:
    co!res; ci?dat;
    ro!dat;
    if
    :: ri?ACT_ACK; res = RES_ACK; goto start;
    :: ri?ACT_NACK; res = RES_NACK; goto start;
    fi }
```

Listing 1.3: Bus example verification

cesses have state variables of type `int` or `intarr` (fixed-size array). No global variables are allowed. The size of `intarr` is implementation-defined, but guarded against overflow. The DSL supports `while`, `if` and `goto` as control flow.

The C backend translates a DSL process to a function and a process call into a function call. The backend assembles all state in a large static `struct`, kept intact between calls. To yield, a process is implemented as a large switch statement, with execution resuming at the most recent label.

From the language subset, Promela generation is straightforward. Each process has two channels: *input* and *output*. **Call**, sends arguments to the callee's input and blocks on the callee's output. Processes block on input until arguments arrive. **Yield** sends the result on the output channel.

Verification properties are specified directly in Promela, exploiting nondeterminism. The complete syntax of our DSL is expressed in Listing 1.4.

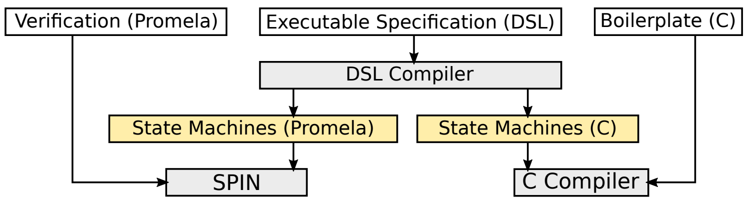

Fig. 4: Workflow of the user provided files (white), intermediary files (yellow), and tools (gray)

4.2 Calculus

We verify our layered system with the following calculus: Each layer has an implementation $\mathit{LayerImpl}_i$, a valid behavior $\mathit{LayerValid}_i$, and a specified behavior $\mathit{LayerSpec}_i$.

$\mathit{LayerSpec}_i$ is a state machine expressed as a function over its past result/action trace returning the next result symbol. $\mathit{LayerSpec}_i$ specifies the correct behavior at layer i.

$\mathit{LayerValid}_i$ is a predicate over the result/action trace that is true if the sequence is permissible at this layer and ends with an action symbol.

$\mathit{LayerImpl}_i$ has the same type as $\mathit{LayerSpec}_i$, except that it must be bound to a state machine of layer $i-1$, which it can query to determine its next step. We denote the binding of this lower level state machine with $\circ$. $\mathit{LayerImpl}_0$ is the exception, which does not need to be bound.

As explained before, isolated specification is not possible, hence they operate on traces that include actions and results for both controller and responder.

In this calculus, our system is verified if it fulfills

$$\forall i.\forall \omega_i.\mathit{LayerValid}_{i+1}\omega_i \Rightarrow \\ \mathit{LayerSpec}_i\omega_i = (\mathit{LayerImpl}_i \circ \mathit{LayerImpl}_{i-1} \circ \ldots \circ \mathit{LayerImpl}_0)\omega_i$$

This verification procedure is depicted in Figure 5 as an infinite sequence of directed messages $\omega = e_1, e_2, \ldots$. We denote the sequence of all messages as ω, and the sequence of messages exchanged between layer i and layer $i+1$ as ω_i. Examples of messages are the action to send an acknowledgement (represented by a down arrow $\downarrow$) or the receipt of an acknowledgment message (depicted as an up arrow $\uparrow$). Even though abstractly we deal simply with a trace of events, it is useful to denote if the message is destined for the controller or responder. We denote this with $c{\uparrow}x$ for a result with value x destined for the controller, and with $r{\uparrow}x$ a result x destined for the responder.

A $\mathit{LayerImpl}_i$ can not only be bound to other implementations, but also to a $\mathit{LayerSpec}_i$. We evaluate the verification time improvements of this in section 6.

We verify this property by encoding it into Promela processes, such that a **verifier** process can not make progress when a violation has been found. Adherence to the protocol is shown by **verifier** always able to make progress. $\mathit{LayerValid}_i$ becomes a non-deterministic process, producing all valid actions. This action is sent to **verifier** which duplicates the action and sends it to both the $\mathit{LayerSpec}$ and the $\mathit{LayerImpl}$. If

```
⟨file⟩        ::= (⟨proc⟩ | ⟨procCopy⟩)* ‘EOF’
⟨procCopy⟩    ::= ‘proccopy’ ⟨id⟩ ‘of’ ⟨id⟩ (‘where’ (⟨rename⟩ (‘,’ ⟨rename⟩)*)? )? ‘;’
⟨proc⟩        ::= ‘proc’ ‘(’ ( ⟨type⟩ (‘,’ ⟨type⟩)* )? ‘)’ ⟨id⟩ ‘(’ ( ⟨varDecl⟩ (‘,’ ⟨varDecl⟩)*
                  )? ‘)’ ‘{’ (⟨varDecl⟩ ‘;’)* ⟨instr⟩* ‘}’
⟨rename⟩      ::= ⟨id⟩ ‘=’ ⟨id⟩
⟨block⟩       ::= ‘{’ ⟨instr⟩* ‘}’
⟨instr⟩       ::= ‘yield’ (⟨aEx⟩ | ‘(’ ⟨aEx⟩ (‘,’ ⟨aEx⟩)+ ‘)’) ‘;’
                |  ⟨varRef⟩ ‘=’ ⟨aEx⟩ ‘;’
                |  ⟨id⟩ ‘:’
                |  ‘while’ ‘(’ ⟨bEx⟩ ‘)’ ⟨block⟩
                |  ‘if’ ‘(’ ⟨bEx⟩ ‘)’ ⟨block⟩ (‘else’ ⟨block⟩)?
                |  ‘goto’ ⟨id⟩ ‘;’
                |  ‘assert’ ‘(’ ⟨bEx⟩ ‘)’ ‘;’
                |  (⟨id⟩ | (‘(’ ⟨id⟩ (‘,’ ⟨id⟩)+ ‘)’)) ‘=’ ⟨id⟩ ‘(’ (⟨aEx⟩ (‘,’ ⟨aEx⟩)*)? ‘)’;
⟨varDecl⟩     ::= ⟨type⟩ ⟨id⟩
⟨type⟩        ::= ‘intarr’ | ‘int’
⟨cEx⟩         ::= ⟨aEx⟩ (‘>=’ | ‘>’ | ‘<=’ | ‘<’ | ‘==’ | ‘!=’) ⟨aEx⟩
⟨bEx⟩         ::= ‘true’ | ‘false’ | ‘(’ ⟨bEx⟩ ‘)’
                |  ⟨bEx⟩ (‘and’ | ‘or’) ⟨bEx⟩ | ⟨cEx⟩
⟨aEx⟩         ::= ‘(’ ⟨aEx⟩ ‘)’ | ⟨varRef⟩ | (‘−’?[‘0’-‘9’]+) | ⟨uOp⟩ ⟨aEx⟩ | ⟨aEx⟩ ⟨bOp⟩ ⟨aEx⟩
⟨bOp⟩         ::= ‘&’ | ‘|’ | ‘*’ | ‘/’ | ‘+’ | ‘−’ | ‘<<’ | ‘>>’
⟨uOp⟩         ::= ‘−’ | ‘+’
⟨varRef⟩      ::= ⟨id⟩(‘[’ ⟨aEx⟩ ‘]’)?
⟨id⟩          ::= ⟨char⟩ (⟨char⟩ | [‘0’-‘9’] | ‘_’)*
⟨char⟩        ::= [‘a’-‘z’] | [‘A’-‘Z’]
```

Listing 1.4: Complete DSL syntax

the *LayerSpec* produces a result, the verifier ensures that the *LayerImpl* produce the same result. If it differs, the verifier will not make progress. We also use this message dispatch to show liveness of the system, by marking it with a SPIN progress label and verifying the liveness check.

We currently do not verify that the layer implementation $\mathit{LayerImpl}_i$ adheres to $\mathit{LayerValid}_i$. Since we verify the full stack of implementations, we still do get the correctness guarantees, but it is possible, that in the middle of the implementation stack, the implementations rely on unspecified behavior.

5 The I²C model

5.1 Layering of I²C

We divide our I²C stack into the layers shown in Figure 6, and we apply the verification principle from section 4 at every layer.

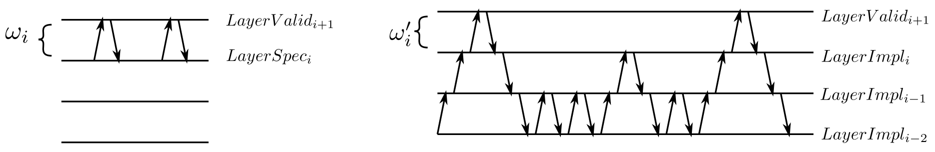

Fig. 5: Verification illustration, the system is verified if $\omega_i = \omega_i'$

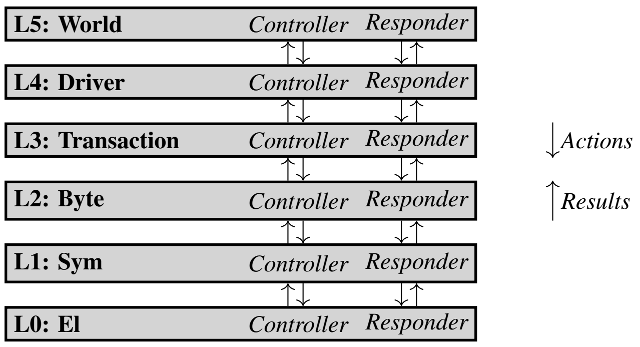
L5: World Controller Responder
L4: Driver Controller Responder
L3: Transaction Controller Responder
L2: Byte Controller Responder
L1: Sym Controller Responder
L0: El Controller Responder
Actions
Results

Fig. 6: Layering of the I^{2}C model.

The stack presented here includes two device-specific layers: World and Driver layer. We envision this process of device modelling and verification to be done for each device that is connected to the bus. This also gives a high level of assurance for the device driver represented by Driver. But it is also feasible to directly interact in a system with, for example, the transaction layer and skip the verification of the higher layers.

5.2 Layer 0: Electrical Layer

The lowest layer, the electrical layer 0, is trusted. Hence it consists of only an *implementation*. It describes how two devices sending bus signals (a SCL/SDA pair, each 0 or 1) are combined into the next bus state, which is sent back to the devices. It does so by using the I^{2}C mandated AND combination of signals for each wire, which is a result of the active drive low logic.

We assume a reliable delivery of bits. Like prior work [5], we observe that I^{2}C bus events can be discretized. We assume sampling of the bus at the Nyquist frequency of the clock, such that two samples occur during a clock high period. This allows us to distinguish START and STOP conditions from BIT0 and BIT1.

5.3 Layer 1: Symbol Layer

Layer interface The symbol layer connects the electrical with the byte layer. It parses a sequence of bits into a symbol and vice-versa, turns a symbol into a bit sequence. The results and actions are depicted in Figure 7. In addition to the I^{2}C symbols we define IDLE and STRETCH, which delay the next symbol.

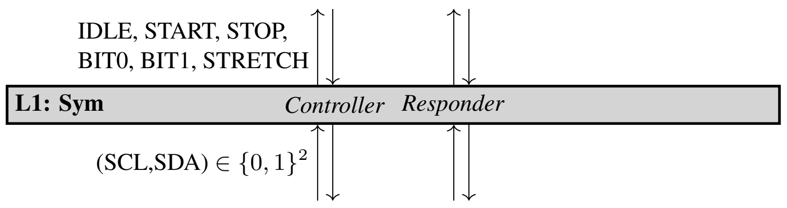

Fig. 7: Interface of the symbol layer. The label describes the datatype of all the channels in this layer.

The implementation differs for controller and responder, but they share a large part of the code (expressed as two sub-processes *SymbolReader* and *SdaDriver*) The controller is actively driving the clock (using a sub-process *SclDriver*). The responder is driving the clock only in one specfic case: When it is processing a STRETCH action, it will delay the clock rise by exactly one cycle. Both controller and responder are clock agnostic. For example the *SdaDriver* will wait until the clock rises and falls again, thus it is invariant against clock stretching. Both byte controller and byte responder are invoked in the same clock cycle. The exception again is during a STRETCH, which will produce an extra invocation in the next clock cycle.

The specification defines how two symbol actions are combined into a new one. In the initial, out-of-transaction state, two IDLE commands are combined into an IDLE result (i.e. $c{\downarrow}$IDLE$, \ldots, r{\downarrow}$IDLE will be followed by $c{\uparrow}$IDLE$, \ldots, r{\uparrow}$IDLE) as well as a START and a IDLE are combined into START (i.e. $c{\downarrow}$IDLE$, \ldots, r{\downarrow}$START will be followed by $c{\uparrow}$START$, \ldots, r{\uparrow}$START).

If a START result has been sent, the specification enters the in-transaction state. In this state, the following action combinations are valid. Note they are symmetrical, thus we skip the sender identifier as well as symmetrical cases.

- ${\downarrow}$BIT1$,{\downarrow}$BIT1 produces two ${\uparrow}$BIT1,
- ${\downarrow}$BIT0$,{\downarrow}$BIT1 produces two ${\uparrow}$BIT0,
- ${\downarrow}$BIT1$,{\downarrow}$START produces two ${\uparrow}$START,
- ${\downarrow}$BIT1$,{\downarrow}$STOP produces two ${\uparrow}$STOP, and enter out-of-transaction state.
- $r{\downarrow}$STRETCH is immediately followed by $r{\uparrow}$STRETCH, until r produces a non stretch action.

The valid actions of the next higher layer follow the same in- and out-transaction states as the specification. Outside a transaction, **ByteValid** either generates an IDLE pair to remain outside a transaction or initiates a transaction by allowing the controller to generate a START. In-transaction it generates a zero or more sequence of STRETCH, followed by all the valid symbol combinations.

5.4 L2: Byte Layer

Layer Interface I^{2}C is a byte-oriented protocol, where each byte is acknowledged. This layer reads and writes symbols, turning them into bytes. START and STOP conditions

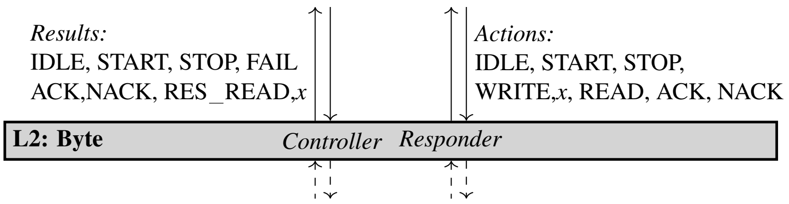

Fig. 8: Interface of the Byte layer. Both controller and responder have the same signature for *actions* and *results*.

are passed through: The higher layer must send START and STOP explicitly. The interface is depicted in Figure 8.

The implementation does not distinguish between controller and responder. START and STOP actions are passed to the symbol layer directly, WRITE,x and READ operate bitwise (MSB first transmitted). If a written bit is not correctly transmitted, the layer will report FAIL and remain silent for the rest of the byte.

The specification describes how actions are combined into results. Controller and responder are not equal anymore; the controller must initiate the transaction. As in the symbol layer specification, an IDLE pair remains outside transaction and a START/IDLE is used to enter the in-transaction state. Within a transaction the following combinations hold

- $c{\downarrow}$ACT_WRITE, x and $r{\downarrow}$ACT_READ is followed by $r{\uparrow}$RES_READ, x, $r{\downarrow}$ACT_(N)ACK and $c{\uparrow}$RES_(N)ACK. Note that the variable x is bound, the written value must be the same as the read value.
- The symmetrical case of the above item where the controller reads and the responder writes.
- ACT_READ can also be combined with ACT_START and ACT_STOP. This is important for the responder, who can not predict the action of the controller, then ACT_READ is a safe choice.

The valid actions follow directly from the specification. All specified combinations are verified, with the caveat that the value of the written byte is constrained to a predefined set of 10 choices. In section 6 we show the trade-offs to verify all values (0 to 255).

5.5 L3: Transaction Layer

I^{2}C defines the transaction format, such that a START condition must be followed by a 7-bit address and a direction bit. Then, depending on the direction bit, the controller reads or writes a sequence of bytes. This introduces an asymmetry: It is the controller that initiates a transaction, and the responder acts accordingly. Figure 9 shows all the actions and events processed at this layer. Starting from this layer, controller and responder have not only distinct specifications as before, but also distinct implementations. The responder also decodes a (currently fixed) I^{2}C address and ignores via NACKs all other addresses.

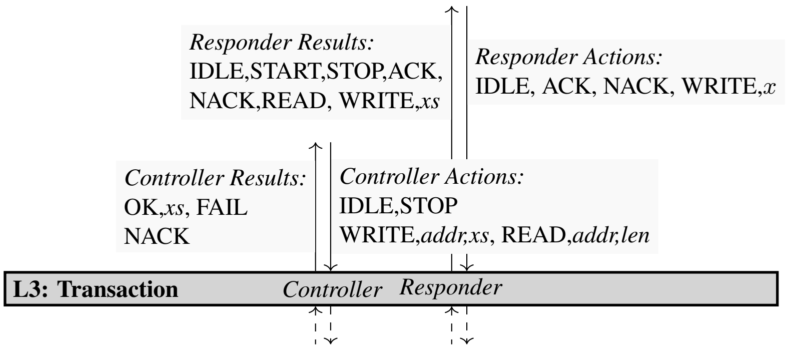

Fig. 9: Interface of the Transaction layer.

The implementation of the controller is fairly straightforward: Each higher level action is turned into a sequence of START and address byte with correct direction bit. If a write is requested, it continues to write the data. If a read is requested, it reads the desired number of bytes, sending an ACK for all except the last byte (per I^{2}C standard) which is answered with NACK that tells the responder to stop sending. If the responder receives a NACK, it is forwarded to the next higher layer.

The responder waits for a START, which is propagated to the driver. We propagate STARTs to the next higher layer to distinguish between two consecutive writes and a write – restart – write sequence. After a START, the responder reads the address byte, checks that the addresses match, and depending on the direction bit propagates either a RES_READ or a RES_WRITE. Since the responder cannot know how many bytes are read, we propagate each byte read request individually to the next level. Writes on the other hand can be buffered, until the controller is done. Then the whole array of written data is passed on.

The specification describes the interaction of driver layer actions. The sequence is determined solely by the controller: If it requests a WRITE of x bytes, we expect a RES_START from the responder, followed by x times a RES_READ. The responder either ACKs $x - 1$ times and then the controller will receive a RES_ACK, or if the responder decides to NACK before, the controller will receive a NACK.

The valid action sequences become conceptually simple but increasingly challenging to verify. The controller produces after a sequence of ACT_IDLE any combination of ACT_READ and ACT_WRITE until finally an ACT_STOP brings it back to the initial, out-of-transaction state. However the data that is either read or written is potentially infinite in length. Since we focused on the correct delivery of data at the lower layer, the verification cases for this layer focus on increasing the length of the transaction. Hence we show that for sequentially increasing payload of any length between 1 and 4 bytes our implementation conforms to the specification.

Conversely, the responder is completely driven by the result it receives. After a RES_START only ACT_IDLE is valid. After receiving START but before a STOP, the following combinations are valid: RES_READ followed by ACT_WRITE,x, RES_WRITE followed by ACT_ACK or ACT_NACK, RES_STOP, and RES_START must be followed by ACT_IDLE.

5.6 L4: Driver

Layer interface At this layer we start implementing the protocol that is specific to our model EEPROM, a Microchip 24XX16 [12]. The controller contains what typically is implemented in the device driver. From the world layer, the controller receives requests for reading or writing from the EEPROM. The responder, on the other hand, encodes the EEPROM-specific features, for example the logic for the address buffer. The responder forwards requests to an EEPROM implementation. The interface is shown in Figure 10.

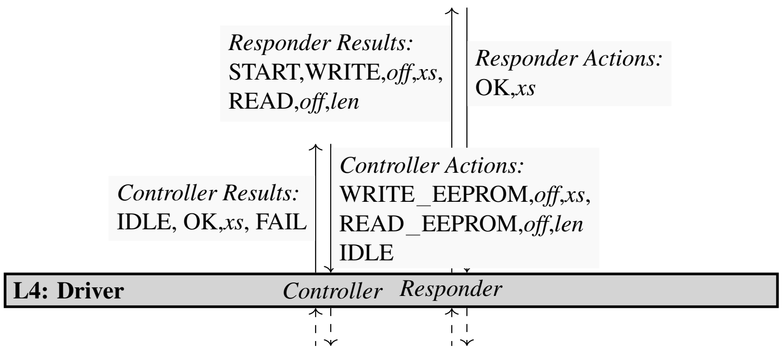

Fig. 10: Interface of the Driver layer.

The implementation of the controller turns an ACT_WRITE_EEPROM, parametrized by an *offset* and a data array, into a long write transaction. The first two bytes determine the EEPROM write offset followed by the data to be written. The data length is not communicated explicitly; if the data is written, the controller sends a STOP condition, signaling to the responder that the transaction is over. Reading works similarly: The controller issues two-byte write transaction followed by a *len*-long read transaction.

The responder behaves similarly. It waits for a START, then expects a write of at least two bytes. If more bytes follow, they are interpreted as a write transaction. It assembles the written data into a buffer and once the STOP condition arrives, passes it on (as RES_WRITE,*xs*) to the world layer. Read is more difficult, because by the time the first read byte must be supplied, the read length is unknown. Hence we assume there is a maximum read length, which we query from World (by issuing a RES_READ) then sending from this buffer.

The specification becomes fairly simple at this point. A controller ACT_WRITE,*off*,*xs* is turned into a responder RES_WRITE,*off*,*xs*. A controller ACT_READ,*off*,*len* becomes a RES_READ,*off*,*maxlen*, followed by a responder ACT_OK,*xs*, and a controller RES_OK,*xs'*, where *xs'* is a prefix of *xs* with length *len*.

The valid action sequence is unconstrained at this point. The controller can choose between ACT_WRITE and ACT_WRITE, while the responder must subsequently receive RES_READ and RES_WRITE and reply with ACT_OK. However, we severely restrict the payload that is transmitted at this level by choosing one of 4 predefined datasets, to keep the full implementation verification time manageable.

5.7 L5: World

Since this is the highest layer, we can not verify the behavior given a higher layer behavior. We still provide a dummy implementation that performs a defined sequence of actions which is what we evaluate on our hardware platform.

6 Evaluation

6.1 Verification runtime

The verification runtimes are evaluated on an AMD Ryzen 9 3900X with 32 GB of RAM running Ubuntu 20.04 with SPIN version 6.4.9.

Verification of the Symbol layer performs a complete state space search and finishes in 0.4 seconds. As mentioned in subsection 5.4, the byte layer is verified only on a small set of payload values, hence we would like to speed up the verification time. We can do so by replacing lower $LayerImpl$ with $LayerSpec$ (e.g. $LayerSpec_{byte} = LayerImpl_{byte} \circ LayerSpec_{sym}$ instead of $LayerSpec_{byte} = LayerImpl_{byte} \circ \ldots \circ LayerImpl_0$).

Table 1 shows the verification times using this method. The speedup factor depends on the layer complexity. Replacing the rather complex **Symbol** with **SymbolSpec** leads to a speedup of $10\times$, replacing **Byte** with **ByteSpec** leads to $5\times$ speedup.

Table 1: Verification time in seconds using different layers of abstraction.

	Full Implementation	SymbolSpec	ByteSpec	TransactionSpec
Symbol	0.1			
Byte	9.0	0.7		
Transaction	69.4	8.7	1.8	
Hl	62.9	6.7	1.0	0.3

Instead of decreasing verification time, this technique can be used to increase the search space size. For example the **Byte** layer can be completely verified (i.e. checking all 256 values for a byte write as well as for a byte read) using **SymbolSpec** in about 70 seconds.

6.2 Execution on a Raspberry Pi

Our DSL can generate C code for the deterministic state machines. Conceptually, the C code can interact with any layer directly. For example, it could get output from the transaction layer and translate it into Linux I^2C API calls. However to profit most from the verification, the whole stack (excluding the electrical layer) can be executed. This leads to an interface that only writes and reads SDA/SCL as high/low states from the bus. We demonstrate this using the Linux GPIO interface connected to an I^2C EEPROM. The boilerplate code runs in an infinite loop: reading the bus state, forwarding it to the controller state machine, reading back the command, and outputting it on the GPIOs. In conformance with the I^2C specification we actively drive the data and clock line low on zero. If a one is to be written, we set the pin to a high impedance state. The process repeats after an appropriate delay accounting for the required setup and hold time.

We currently hardcode the testcase in the World implementation. To expose an interface, we would also replace the highest layer with boilerplate C code that would e.g. do non-blocking reads from a UNIX pipe do receive the commands to be sent.

The total required boilerplate code consists of 128 lines of code; most of it implements interfacing with Linux's file-based `sysfs` GPIO interface. The generated I^2C state machine code consists of 2678 lines and compiles to a binary of 32 KB.

7 Conclusion and Future work

We have successfully demonstrated that our approach of creating and verifying layered specifications for I^2C is feasible, and that it can be used to express bus interactions on a high level, to specify the expected behavior, and to verify that the specification fulfills this property. Furthermore, the executable specification can be used to generate code, which interacts with real physical devices.

While this work has already proven to verify desirable properties in a specific scenario, we cannot yet claim full generalization. We have empirically considered partially-conforming devices, but we did not formally model them at this stage. We expect the specification to be extended, but no change in the methodology nor the lower layers should be necessary. I^2C features we do not yet handle include broadcast, variable length read transactions and multi controller.

So far, we generate C code. In future work, we plan to generate synthesizable hardware descriptions, producing verified FPGA or even ASIC implementations. While we do not expect any problems with the state machine generation itself, we will also need to generate and verify the corresponding hardware-software interface.

From a theoretical perspective, we so far assumed that our layer calculus itself is correct, i.e. we assumed that a layer that follows the specification can be combined with any (lower and higher) layer that also fulfills the layer contract. Given the higher order nature of this, we think that either an embedding in an existing process calculus or a from-scratch formalization in an interactive theorem prover would be interesting. The second option would also open the possibility to reason about the system not only in a model checker, but in a theorem prover. This could lift the restriction that we verify only on a small set of payloads, at the cost of some manual proof engineering.

References

1. Raspberry pi i2c clock-stretching bug. `https://www.advamation.com/knowhow/raspberrypi/rpi-i2c-bug.html`. Accessed: 2021-04-01.
2. Video capture driver (video for linux 1/2). `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/media/i2c/ks0127.c?h=v5.8.3`. Accessed: 2021-04-01. Unfortunately the datasheet is not public.
3. ams AG. *AS5011 Low power Integrated Hall IC for human interface applications*, 2009. Rev. 3.6.
4. Bernard Boigelot and Patrice Godefroid. Model checking in practice: An analysis of the access.bus^{TM} protocol using spin. In Marie-Claude Gaudel and James Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 465–478, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
5. S.H.J. Bos and M.A. Reniers. The I2C-bus in discrete-time process algebra. *Science of Computer Programming*, 29(1-2):235–258, July 1997.
6. Dragan Bošnački, Aad Mathijssen, and Yaroslav S Usenko. Behavioural analysis of an i2c linux driver. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 205–206. Springer, 2009.
7. Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking hyperltl and hyperctl*. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 30–48, Cham, 2015. Springer International Publishing.
8. Saurav Gorai, Saptarshi Biswas, Lovleen Bhatia, Praveen Tiwari, and Raj S. Mitra. Directed-simulation assisted formal verification of serial protocol and bridge. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, page 731736, New York, NY, USA, 2006. Association for Computing Machinery.
9. Gerard J Holzmann and William Slattery Lieberman. *Design and validation of computer protocols*, volume 512. Prentice hall Englewood Cliffs, 1991.
10. Ke Jiang and Bengt Jonsson. Using spin to model check concurrent algorithms, using a translation from c to promela. In *MCC 2009*, pages 67–69. Department of Information Technology, Uppsala University, 2009.
11. Arjen Klomp, Herman W Roebbers, Ruud Derwig, and Leon Bouwmeester. Designing a Mathematically Verified I2C Device Driver Using ASD. In *CPA*, pages 105–116, 2009.
12. Microchip. *24XX16: 16K I2C Serial EEPROM*, 2019.
13. NXP Semiconductors. *I2C-bus specification and user manual*, 4 2014. Rev. 6.
14. ON Semiconductor. *CAT5259 Quad DigitalPotentiometer (POT) with 256 Tapsand I2C Interface*, 2013. Rev. 11.
15. Can Pan, Jian Guo, Longfei Zhu, Jianqi Shi, Huibiao Zhu, and Xinyun Zhou. Modeling and Verification of CAN Bus with Application Layer using UPPAAL. *Electronic Notes in Theoretical Computer Science*, 309:31–49, December 2014.
16. A. Roychoudhury, T. Mitra, and S. R. Karri. Using formal techniques to debug the amba system-on-chip bus protocol. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 828–833, 2003.