ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 463

Systems Group, Department of Computer Science, ETH Zurich

Generating Trustworthy I^2C Stacks Across Software and Hardware

by

Zikai Liu

Supervised by

Prof. Dr. Timothy Roscoe
Daniel Schwyn

March 2023–September 2023

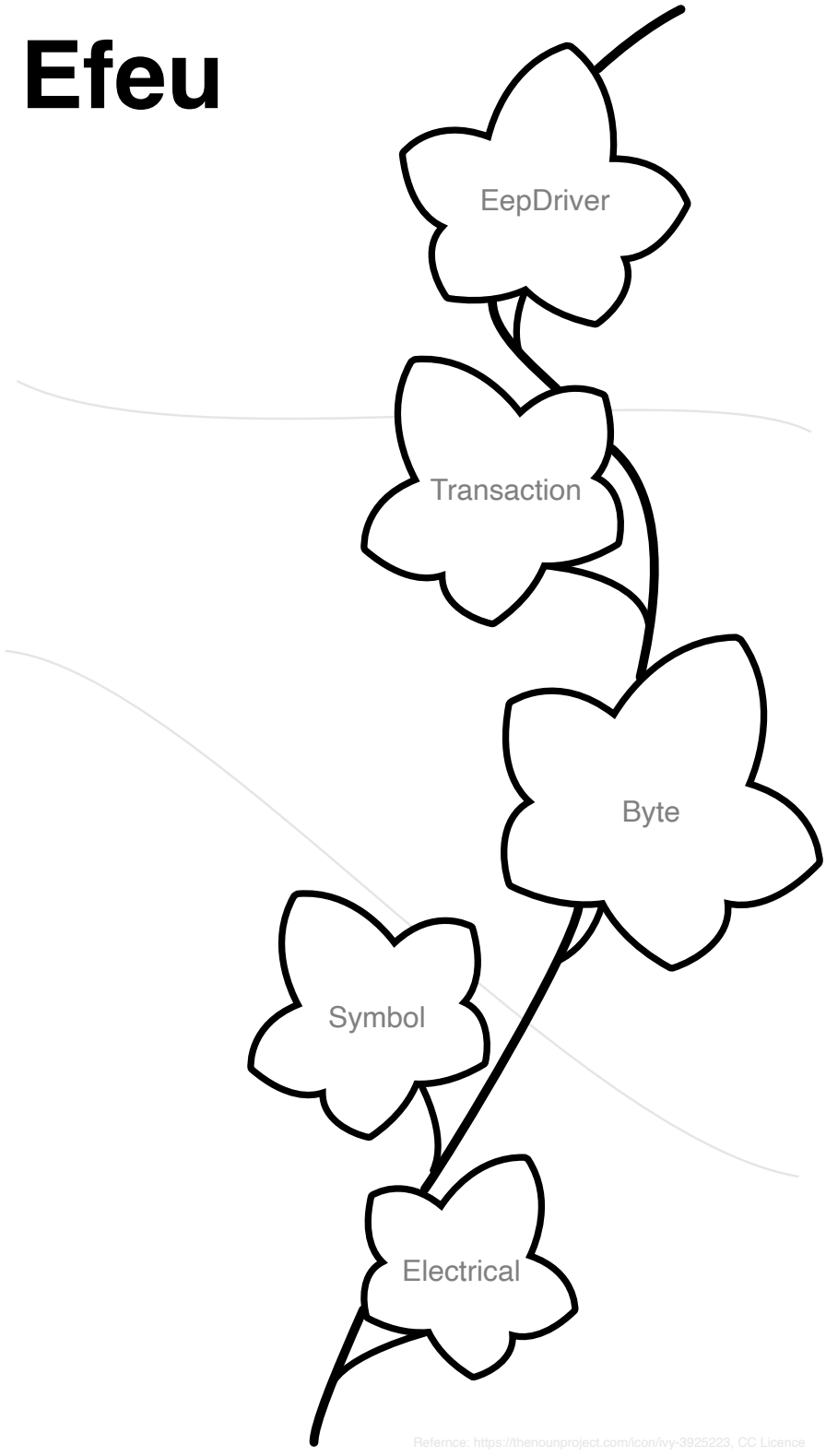
D INFK

Abstract

Modern computers increasingly offload low-level management to baseboard management controllers (BMCs). While the research community is actively working toward a trustworthy BMC firmware stack, writing efficient and highly-assured device drivers remains a challenge. One such example is creating I^2C drivers to accommodate devices that implement non-compatible variations of the communication protocol. Prior research proposes a model-checked I^2C specification, from which device drivers can be generated. While the viability of this approach has been demonstrated, there remain several limitations preventing the generated drivers from being useful in real-world applications.

In this work, we present Efeu, a framework to synthesize assured and practical drivers from model-checked specifications. Advancing beyond the previous effort, Efeu can generate combined software-hardware drivers, with users freely selecting split points between components and Efeu handling the interface generation. Evaluations on real devices validate the generated I^2C stacks. Compared with manually crafted drivers, the generated stacks have a reasonable overhead in performance and resource utilization, while featuring the assurance gained from model checking.

Efeu



Contents

Chapter 1

Introduction

With increasing complexity in system design, modern computers offload low-level management tasks to a dedicated processor, the Baseboard Management Controller (BMC). Behind the scene, BMCs control critical parts of the platform, including power distribution to the rest of the system. Needless to say, BMCs act as a cornerstone for functional correctness and security assurance of the whole system. Buggy or misconfigured BMCs can render the system inoperative or even cause irreversible hardware damage.

Despite its importance, existing BMC firmware lacks assurance. As of this writing, there are 221 BMC-related CVE records[1]. The research community is actively working toward a trustworthy BMC stack. One notable approach is building the firmware on top of formally verified operating systems (OSs) like seL4 [3, 11, 19, 25].

While the highly-assured OS provides a robust base, several building blocks are still missing toward a complete BMC firmware stack, notably device drivers. Being unable to communicate with its peripherals essentially renders a BMC useless. Unlike Linux-based BMC stacks like OpenBMC [5], available device drivers on seL4 are yet limited.

While writing drivers is a routine task, writing drivers *properly* remains a challenge. For instance, I^2C is a protocol being widely adopted for communication between components on the baseboard, such as voltage regulators. Despite its significance, I^2C is defined in ambiguous informal English-language documents [26]. Various devices implement non-standard variations of the protocol. For example, while the standard specifies the read flag as an indication of a read transaction, in which subsequent payloads are supplied by the I^2C responder, the AS5011 Hall Sensor [2] assigns a different meaning to the flag. For that device, the read flag does not indicate a read *I^2C transaction*,

[1] CVE record search (`https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=BMC`), accessed 2023-09-11.

but a *read operation to its register*. The first byte of the payload—register index, is supplied by the controller rather than the responder. This protocol violates the I^2C standard at the transaction level. In addition to AS5011, there are a handful of devices violating the standard at various levels [14].

To address these challenges, previous works [14, 15, 27] propose applying model checking on an I^2C driver specification and subsequently generating drivers from it. While these studies demonstrate the feasibility of this approach, there remain several limitations preventing the generated drivers from being useful in real-world applications.

Built upon the prior research, in this work, we introduce Efeu, a framework to design drivers for model checking and software/hardware implementation. Advancing beyond the previous effort, Efeu can generate combined software-hardware I^2C stacks. Users may choose the split point between software and hardware, and even generate drivers that cross the software-hardware boundary multiple times. Several limitations in the previous works are addressed to improve the usability of the generated drivers. Furthermore, We conduct tests on real-world platforms. Evaluations show that the generated stacks function correctly with a reasonable overhead on performance and resource utilization compared to manually crafted drivers, while featuring the assurance from model checking.

While several studies have explored approaches of modeling and synthesizing drivers [4, 22, 23, 24, 28, 31], these works mainly focus on leveraging the domain-specific languages (DSLs) to enforce abstractions and therefore improve the driver reliability. In contrast, Efeu employs a C-like DSL, which we believe is more user-friendly and lowers the barrier for using the framework, while the driver assurance is achieved with model checking—extensive search through the state space. Furthermore, to our knowledge, Efeu is the first framework that enables flexible compositions of hardware and software components into a single I^2C stack.

The remainder of this thesis is organized as follows. Chapter 2 introduces background material. In Chapter 3, we describe the design of Efeu, including its programming model and languages. Chapter 4 further discusses the Efeu compiler in detail. After implementing the system, we performed evaluations on the I^2C driver stack as a demonstration of Efeu's functionalities. Chapter 5 presents the model checking results, followed by our evaluation with real-world devices in Chapter 6. In Chapter 7, we discuss a few other aspects of our system, including its further applications on BMC firmware and future work. Finally, Chapter 8 offers our conclusions.

Chapter 2

Background

In this chapter, we present the background pertinent to this work. The framework we present touches upon various areas in systems research and engineering, involving electrical engineering, FPGA-based hardware design, High-Level Synthesis (HLS), software development, and model checking.

2.1 I^2C

I^2C is a low-speed serial communication protocol [26]. It is widely adopted for connecting microcontroller and peripheral devices in embedded systems. The protocol operates over two wires, a serial data line (*SDA*) and a serial clock line (*SCL*), both pulled up by external pull-up resistors. Connected devices may only drive the lines low.

Designed with the principle of simplicity and versatility, I^2C allows multiple devices to communicate over the same bus. Each device is identified by a 7-bit address. An I^2C device is either a *controller* (master) or a *responder* (slave). Controllers initialize and control the data transfers. Only controllers drive the clock line (except clock stretching discussed below), while responders react accordingly.

2.1.1 Electrical Behavior and Symbols

I^2C uses the two wires to encode bits (0 or 1) as well as the start and stop signals of bus transactions. Figure 2.1 shows the encodings of the four symbols on I^2C buses. START and STOP symbols are produced by controllers. BIT0 and 1 are generated either by controllers (during a write) or by responders (during a read). To generate BIT0, the SDA is held low when SCL is high. For BIT1, the device releases the bus (SDA is pulled high externally). Devices should only sample the SDA line when SCL is

high. Level transitions on SDA should only happen while SCL is low (except START and STOP).

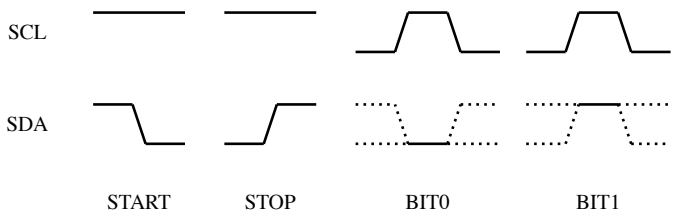


Figure 2.1: I²C symbols.

The I²C protocol defines multiple speeds. Standard Mode has a maximum bit rate of 100 kbit/s. The Fast Mode supports up to 400 kbit/s. Additionally, the standard defines Fast Mode Plus (1Mbit/s) and High Speed Mode (3.4Mbit/s), which change many aspects of how the bus operates and are therefore not covered in this work. Note that these speed modes define the maximum speeds, not guaranteed rates. SCL is driven by controllers but not responders (except clock stretching). Therefore, it is the controller that decides the actual transmission rate. If a controller operates at a reduced speed, compatible devices should still function.

2.1.2 Byte Format

Bytes are encoded with BIT 0 and 1 symbols and must consist of 8 bits. The Most Significant Bit (MSB) is sent first. Each byte, if received by the target device, shall be Acknowledged (*ACK*) after the last bit. In that clock cycle, the controller releases the SDA line, and the target device, if exists and acknowledges the byte, shall drive the SDA low. If the target device does not exist on the bus, no device pulls down SDA, which results in SDA being unaltered as a Not Acknowledged (*NACK*).

2.1.3 Transactions

Built on top of the byte format, the standard defines *transactions*. A transaction is composed of one or more *messages*. Each message starts with a START symbol, succeeded by a 7-bit target address plus a 1-bit read/write flag (which makes up a byte and should be ACKed or NACKed). Subsequent to this, the message payload is transmitted byte by byte. Each byte is ACKed or NACKed respectively. A transaction concludes with a STOP symbol. For transactions comprising only one message, it has the following content, where `N` is the number of bytes in the payload.

```
START, 7-bit address + read/write bit, ACK/NACK, {payload byte,
    ACK/NACK} * N, STOP
```

When a controller opts to send and/or receive multiple messages in a transaction, it can proceed with the next message immediately after the one before, without issuing a STOP. The STOP is only sent after the last message to conclude the whole transaction. From the second message onward, the START symbols are called *restart* symbols. This approach keeps the bus busy for the whole transaction. Given that each message contains its 7-bit address, messages in one such transaction can be directed to different target devices. In this scenario, a transaction is structured as follows, with `M` denoting the number of messages within the transaction.

```
{START, 7-bit address + read/write bit, ACK/NACK, {payload byte,
    ACK/NACK} * N} * M, STOP
```

2.1.4 Clock Stretching

Typically, only controllers can drive SCL. However, when a responder fails to prepare data for a transmission in time, it may signal the controller by clock stretching. Clock stretching pauses the I^2C transaction by holding the SCL low. In essence, it pauses the clock and the controller can no longer raise SCL. Devices on the bus should sense the stretching and remain in standby mode until SCL is released.

2.1.5 Arbitration

When multiple controllers are connected to the bus and initiate transactions simultaneously, *arbitration* determines which transaction wins. SCL and SDA are pull-up lines. If both BIT 0 and BIT 1 are transmitted at the same time, the bus registers BIT 0. Devices, while sending bits on SDA, also read back the level. Transmitting a BIT 0 can never fail given the electrical operations are applied correctly. Transmitting a BIT 1 may fail, if another device is sending BIT 0 at the same time, resulting in SDA being pulled down. Here, the device sending BIT 1 loses the arbitration and should terminate its transaction. Prior bits sent do not interfere with the other message since they have exactly the same bits, otherwise the conflict would arise earlier. The other device is unaware of the arbitration process, as all its symbols are transmitted without issues. In a rare case where two messages are exactly identical, they are both transmitted correctly, but the target device receives a single copy.

Arbitration is relevant only if multiple *controllers* are present in the system. Responders are not involved in the arbitration process. If there is a single controller on the bus, we can safely ignore the arbitration process (there are exceptions like SMBus ARA [9], where arbitration happens among responders that pull down the alert line).

2.1.6 Non-Compliant Devices

Several devices violate the I^{2}C standard. Humbel's work [14] identifies and elaborates a few real-world devices that are not compliant to the standard. In this subsection, we revisit one such device that is pertinent to our evaluation.

AS5011 Hall Sensor In the I^{2}C standard, the bit succeeding the 7-bit address indicates whether the transaction is a read or a write. In a read transaction, the subsequent bytes should be supplied by the responder. However, the AS5011 Hall Sensor [2] violates this. The protocol implemented by this sensor uses the read/write bit to indicate read/write to the *device register*. In either case, the first byte of the payload is the register index being always supplied by the controller. Only the data that follows respect the read/write bit.

This device violates the standard Transaction layer as a read transaction incorporates a byte to write. A specialized Transaction layer is written for it, but the standard layers below are reused.

2.2 SPIN and Promela

SPIN is a software tool that enables the formal verification of concurrent and distributed systems. It was first developed in the 1980s and has since become a widely used tool for software verification. It works by taking a finite-state model of a system and exhaustively exploring all possible states of the model to check for the presence (or absence) of errors or violations of specified properties, such as common problems in concurrent systems like deadlocks and livelocks, as well as properties specified by the user.

Promela, short for Protocol/Process Meta Language, is the modeling language used by SPIN. It allows the user to describe a system's behavior as a set of processes running concurrently and communicating through message passing. Processes can be created dynamically. Message passing can be synchronous or asynchronous.

One of the powerful features of Promela/SPIN is its capability to model non-deterministic behaviors with ease. For example, Figure 2.2 shows an example of a non-deterministic `do` loop in Promela. When this inline function is invoked, it non-deterministically outputs 0 to infinite `SYM_STRETCH`s to the channel. This feature allows modeling large state spaces with relatively low effort rather than specifying all combinations.

SPIN/Promela is used for model checking the I^{2}C stack in the previous works [14, 15, 27] as well as this work. We discuss the I^{2}C stack and the Promela verifiers being used in Section 2.3, and our verification result in Chapter 5.

```
1  mtype = {SYM_STRETCH};
2
3  chan channel = [0] of {mtype};
4
5  inline stretch(){
6      do
7      :: channel ! SYM_STRETCH;
8      :: break;
9      od
10 }
```

Figure 2.2: An example of non-deterministic do loop in Promela.

2.2.1 Non-Progress Cycle Check

Non-progress cycle (NPC) checks are used to detect livelocks in the system being modeled. In SPIN, NPC check [10] requires the user to mark the desired activities of the system in the process with a `progress` label. With the check enabled when compiling the verifier, SPIN tries to find a cycle that does not involve the progress label. If no such cycle exists, the system definitely makes progress eventually.

2.3 Model-Checked I^{2}C Stack

As discussed in Section 2.1, the I^{2}C protocol, while widely adopted, is plagued with the presence of numerous incompatible devices that violate the specification at various abstraction levels. To simplify the development of highly assured drivers, Humbel *et al.*propose a framework [14, 15] that enables rapid modeling of non-compliant devices, applying model checking, and generating drivers. Specifications are written in a Domain-Specific Language (DSL), from which both Promela code and C code are generated. The Promela code is used for model checking with SPIN. The C code is used to construct software drivers.

Using the framework, the I^{2}C stack is modeled in layers. Layers can be reused to model non-compliant devices by simply replacing the standard-violating layers while keeping the others unchanged. To demonstrate the real-world usage of the framework, the work focuses on modeling an EEPROM driver stack.

Built upon the framework, Störzbach's work [27] extends it by introducing a backend to generate hardware drivers. It takes the same I^{2}C specification and generates VHDL (a hardware description language). The hardware driver can be implemented with EDA tools like Xilinx Vivado and run on FPGAs (Field Programmable Gate Arrays).

This work, as an extension and generalization of the previous works, focuses on the same I^{2}C stack of the EEPROM driver. In this section, we discuss the I^{2}C stack and its specification in the previous works.

2.3.1 Layered I^{2}C Specifications

In previous works [14, 15, 27], the specification models both a I^{2}C controller stack and a I^{2}C -based EEPROM responder stack. Both stacks consist of the layers described below. In this work, while we use both stacks for model checking and software simulators, our primary emphasis is the *the controller stack*. We use it as an example to illustrate the framework's efficacy in modeling and implementing drivers for real-world applications. Consequently, we revisit the controller stack in detail below, while omitting the responder stack for brevity. To see the complete specification of the responder stack, readers are directed to the previous works [14, 15, 27]. Alternatively, Appendix A shows the complete specification for both stacks using ESI, the new topology specification language in our framework (discussed in Section 3.5).

We describe layers from the lowest-level layer—the Electrical layer, to the highest-level layer—the World layer. Each layer is described with one or more paragraphs below. Between these paragraphs, we use code blocks to describe the interface in between. ↑ and ↓ denote the communication direction. For example, between **Electrical Layer** and **Symbol Layer** below, the ↑ code block in between specifies the interface from Symbol to Electrical, and vice versa.

Electrical Layer The lowest layer models the I^{2}C wires: SCL and SDA. It describes how levels on the same bus are combined into the next bus status. For example, given that SCL and SDA are pulled-up wires, a 0 (LOW) and a 1 (HIGH) from two devices to one of the wires results in 0 (LOW) as the next status.

↑

- SCL to send: 0, 1.
- SDA to send: 0, 1.

↓

- Actual SCL on the bus: 0, 1.
- Actual SDA on the bus: 0, 1.

↑ ↓

Symbol Layer This layer is responsible for conversion between I^{2}C symbols (START, STOP, BIT0, and BIT1) and electrical levels. In addition to the symbols defined by the I^{2}C standard, two more symbols, IDLE and STRETCH, are defined. IDLE puts the bus in the idle state for one more cycle. STRETCH stretches SCL for one cycle.

↑ ↓

- Symbol to send: IDLE, START, STOP, BIT0, BIT1, STRETCH.
- Symbol received: IDLE, START, STOP, BIT0, BIT1, STRETCH.

↑ ↓

Byte Layer The I^2C protocol is byte-oriented (7-bit address + r/w bit also forms a byte). This layer handles encoding/decoding bytes to/from bits as well as ACK/NACK of each byte.

↑ ↓

- Action to Byte layer: BYTE_ACT_START, BYTE_ACT_STOP, BYTE_ACT_WRITE, BYTE_ACT_READ, BYTE_ACT_ACK, BYTE_ACT_NACK, BYTE_ACT_IDLE.
- Byte to send (only valid for BYTE_ACT_WRITE).
- Result from Byte layer: BYTE_RES_OK, BYTE_RES_ACK, BYTE_RES_NACK, BYTE_RES_READ, BYTE_RES_START, BYTE_RES_STOP, BYTE_RES_IDLE, BYTE_RES_ARB_LOSS.
- Byte received (only valid for BYTE_RES_READ).

↑ ↓

Transaction Layer The Transaction layer encodes messages into bytes and decodes bytes into messages. A message starts with a START symbol. However, it does not necessarily end with a STOP symbol, but is followed by a restart message. The transaction only terminates with a STOP succeeding the final message. Consequently, available actions to the Transaction layer not only include TR_ACT_READ and TR_ACT_WRITE that automatically send the heading START, but also a TR_ACT_STOP action that sends the STOP symbol. The payload of each message may have variable lengths, but the interface must be in a fixed size. The specification in previous works defines an array of size 16 to pass the payload as well as a length field. We stick to the same configuration.

↑

- Action to Transaction layer: TR_ACT_WRITE, TR_ACT_READ, TR_ACT_STOP, TR_ACT_IDLE.
- 7-bit address.
- Payload length.
- Payload data: 16 bytes array.

↓

- Result from Transaction layer: TR_RES_OK, TR_RES_FAIL, TR_RES_NACK.
- Payload length.
- Payload data: 16 bytes array.

EepDriver Layer At this layer, we implement the byte-addressable EEPROM (Electrically Erasable Programmable Read-Only Memory) communication protocol [20]. The test device is a Microchip 24AA512. To read data from EEPROM, the two-byte data address is first written on the I^2C bus. Subsequently, bytes are read one by one. The address is automatically incremented by one after every byte by the EEPROM. Sending the data address (not the I^2C address of the EEPROM) and reading the data should happen in one transaction. Similarly, to write data to the EEPROM, the address is written on the I^2C bus, followed by bytes to write. The interfaces between this layer and the upper layer are similar to those with the Transaction layer, except that the result from EepDriver does not contain the payload length, since the higher level should know the number of bytes read or written.

↑

- Action to EepDriver layer: EEP_ACT_WRITE, EEP_ACT_READ.
- 7-bit address.
- Payload length.
- Payload data: 16 bytes array.

↓

- Result from EepDriver layer: EEP_RES_IDLE, EEP_RES_OK, EEP_RES_FAIL, EEP_RES_NACK.
- Payload data: 16 bytes array.

World Layer The World layer is the highest layer in the stack. In the original works, this layer performs performs a fixed sequence of actions indefinitely. In the new framework we present, this layer can not only perform fixed actions as before, but can also serve as an entry point for user-specified inputs, depending on the configuration.

2.3.2 DSL

The DSL in the previous works [14, 15, 27] has a syntax close to C, but is neither a subset nor a superset. In this subsection, we discuss the important features of the DSL that are relevant to this work. For the complete descriptions of the DSL, readers are directed to the original works.

```
1  proc (int, int) UpperLayer (int valueFromLayer) { /* some code
       */ }
2
3  proc (int, int, int, int) Layer (int valueFromLowerLayer,
4                      int anotherValueFromLowerLayer) {
5    int a;
6    int b;
7    /* ... */
8
9    yield (1, 2, 3, 4);
10   /* valueFromLowerLayer and anotherValueFromLowerLayer are
        changed */
11
12   (a, b) = call(42);
13 }
14
15 proc (int) LowerLayer (int valueFromEvenLowerLayer) { /*
       LowerLayer */ }
16
17 system {
18   layers [Bottom, Middle, Top];
19   device Dev {
20     Bottom : LowerLayer;
21     Middle : Layer;
22     Top : UpperLayer;
23   };
24 }
```

Figure 2.3: Example of the original DSL.

Layers (Processes) Layers are defined with coroutines (processes). The syntax is similar to C functions with a few changes. Firstly, instead of returning a single value, a layer can "return" a tuple of values. More precisely, those are values that are `yield`ed (discussed below) to the lower layer. Unlike C functions, layers are ever-running processes that never "return". Similarly, the function parameters are values passed from the lower layer to the upper layers. For example, in Figure 2.3, on every communication, `Layer` passes four integers to its lower layer and receives two integers in return. *Each layer implicitly get a set of values passed from the lower layer on entry.*

`yield` The `yield` operation communicates with the *lower* layer from the current layer. As its name suggests, it passes values to the lower layer, waits for the lower layer to finish processing, gets back values, and resumes the execution right after the `yield`. Note that the values of the "function arguments" of the current layer (`valFromDownstream` above) are changed after `yield`, although it seems like the execution just continues. The usage of `yield` is determined given the layer definition. Line 9 in Figure 2.3 shows a valid `yield` operation.

`call` The `call` operation communicates with the *upper* layer from the current layer. The values to the upper layer are passed as arguments, and the values `yielded` by the upper level is assigned to local variables. Line 12 in Figure 2.3 shows a valid `call` operation. Notice that the usage of `call` is *not* determined by the signature of the current layer, but that of the *upper layer*. Only after knowing what the upper layer is can the usage of `call` be determined. An interface mismatch results in a compile time error [15].

Assembling Layers As shown in Line 17 to 24 in Figure 2.3, layers are assembled in the `system` block. `layers` defines the names of layers in order. `device` defines a new device by specifying the process for each layer.

Types The original DSL only allows 32-bit signed integer (`int`) and fixed-size `int` array `intarr` with size 16.

2.3.3 Verifiers

To perform model checking on the I^{2}C stack, in addition to the Promela code generated from the DSL, extra Promela code written manually is needed.

The previous works by Humbel [14, 15] include several verifiers to check various functionalities of the I^{2}C stack. An important part is equivalence checking. Each of those verifiers assembles both a controller and a responder up to a certain layer. The composition is then driven with all valid inputs that can be emitted by the upper layers. The same inputs are fed into an abstract machine that specifies the expected output for such a composition. If the outputs diverge, the equivalence checking fails. Figure 2.4 illustrates the architecture of a verifier that verifies layers up to the Byte layers.

Be careful about the claims made by the verifiers. For the verifier shown in Figure 2.4, it does *not* claim either the controller Byte and Symbol or the responder Byte and Symbol conforming to the spec. Instead, it claims *the composition* conforms to the spec. Consider the following case: if the encoding of the START symbol is changed in the Symbol layers of both the controller and the responder but still agree with each other, the composition still aligns with the spec from the viewpoint of the upper level—the Transaction layer.

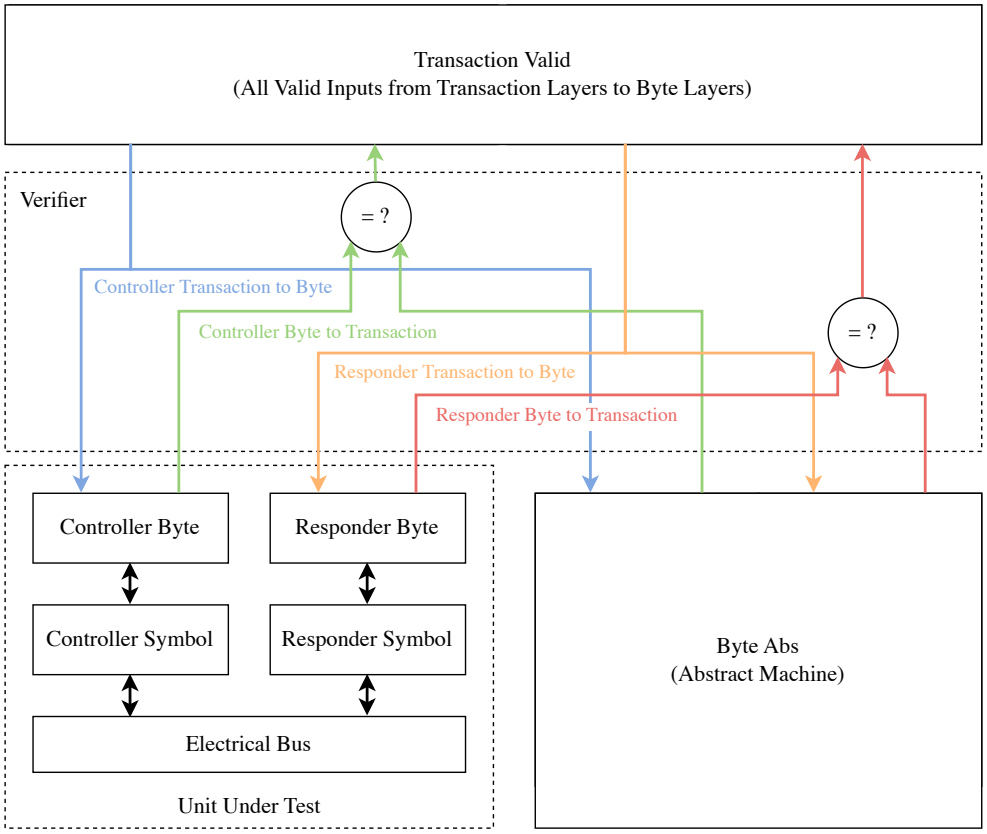


Figure 2.4: Verifier for equivalence checking on byte layers.

In the evaluation of the framework discussed in this work, we use the equivalence checking verifiers up to Symbol, Byte, Transaction, and EepDriver. There are a few verifiers that are not maintained in the codebase of the previous works. Due to the time constraint, we do not port them. Porting them would be future work.

2.3.4 Generating Software Drivers

Humbel's compiler [14, 15] translates the DSL to C code. Given the similar syntax of the DSL as C, the majority of these translations are direct and intuitive. We discuss the most notable parts here. For the complete descriptions of the translation rules, readers are directed to the original works. Figure 2.5 shows the generated C code from the DSL example in Figure 2.3.

Processes to C Functions Data fields originating from the downstream layer are translated into passed-by-value function parameters. Conversely, fields directed towards the downstream layer (i.e. values being yielded) are translated into pass-by-reference parameters. Line 10 and Lines 14–16 in

Figure 2.5 show the translated C function signatures from `UpperLayer` and `Layer` in Figure 2.3. Note that when a resulting C function is called for the first time, function arguments carry values from the downstream layer. This preserves the semantics that layer get a set of values from the lower layer on entry.

`call`s to C Function Calls Each `call` in the DSL is translated to a C function call. The parameters are supplied as passed-by-value arguments. The variables accepting the returned values are supplied as pass-by-reference arguments. It aligns with the C function signature being invoked. Line 34 in Figure 2.5 shows the translated C function call from Line 12 in Figure 2.3.

`yield`s to Continuations Each `yield` in the DSL is translated into a *continuation* in the resulting C function. Continuations are a technique to create the illusion that execution resumes at a point, semantically similar to function calls. However, instead of creating a stack frame, it tears down a layer of stack frame (return). The process involves the following steps.

1. Assign values to output variables by dereferencing the return parameters (Line 26–29 in Figure 2.5).
2. Set a unique (in the current function) identifier to the global variable `mapos` (Line 27)
3. Return (Line 28).
4. Insert a label right after the return (Line 29).
5. In the big switch at the beginning of the function, on that `mapos` value, jump to the label (Line 17–19).

Essentially, this approach saves the instruction pointer right after the `yield` to static region but uses labels and `goto` constructions in C.

Local Variables to (Global) Static Variables Since `yield`s involve re-entering the function, all on-stack local variables must be transformed to static global variables to preserve their values across function calls. With the original compiler, the procedure is done by including those variables in the global data structure (Line 3–4 in Figure 2.5).

2.3.5 Generating Hardware Drivers

Störzbach's work [27] extends Humbel's work [14, 15] by adding a backend to the compiler that generates VHDL source code for hardware implementation. The translation happens at the source code level. We discuss the relevant constructions of this backend in this subsection. For the complete translation rules, please refer to Störzbach's work [27].

```
1  static struct {
2    struct {
3      int a;
4      int b;
5      int mapos;
6    } Dev_Middle;
7    /* Dev_Top and Dev_Bottom */
8  } global;
9
10 void Dev_Top(int valueFromLayer, int *out_0, int *out_1) {
11   /* some code */
12 }
13
14 void Dev_Middle(int valueFromLowerLayer,
15                 int anotherValueFromLowerLayer,
16                 int *out_0, int *out_1, int *out_2, int *out_3) {
17   switch (global.Dev_Middle.mapos) {
18   case 0:
19     goto continuation_0;
20   case 1:
21     goto continuation_1;
22   };
23 continuation_0:
24   /* ... */
25
26   *out_0 = 1;
27   *out_1 = 2;
28   *out_2 = 3;
29   *out_3 = 4;
30   global.Dev_Middle.mapos = 1;
31   return;
32 continuation_1:
33
34   Dev_Top(42, &global.Dev_Middle.a, &global.Dev_Middle.b);
35 }
36
37 /* Dev_Bottom */
```

Figure 2.5: Generated C code from the DSL example in Figure 2.3.

Types Signed 32-bit integer and 16-slot fixed-size array are the only two types defined by the DSL. The former is mapped to `signed(31 downto 0)` in VHDL. The latter is mapped to a defined type `int_array`, which is `array(15 downto 0)of signed(31 downto 0)`.

Interfaces Each layer in the DSL is translated to a VHDL entity. Communications between layers adopt the handshaking protocol (discussed in Subsection 2.5.1). For each communication channel from one layer to another, there are one or several data ports plus a valid pin and a ready pin. Each layer is connected to an upper layer and a lower layer, and connections are

bidirectional. Therefore, there are four sets of data plus valid and ready ports per layer. In addition, each entity has a clock input and an active-low reset input.

VHDL Processes for Sequential Code The main part of the VHDL entity for the layer is a VHDL process. Variables in the DSL are translated to process variables. Sequential code in the DSL are translated to sequential assignments in VHDL (except input/output of `yield`s and `call`s). Arithmetic operations as well as control flow operators allowed in the DSL all have equivalences in VHDL.

Finite State Machines Each label in the DSL, including the ones inserted by the compiler for the entry block and `yield`s/`call`s [27], are translated to a state in VHDL. A `goto` statement becomes an assignment to the state variable. On the next cycle, the state machine executes logic in the target state, simulating the `goto` process.

State Transfer of `yield` In the state machine of every layer, three states dedicated for `yield`s are inserted. Figure 2.6 shows the state transfer of a `yield` operation. All `yield` share the same intermediate states `READY_WAIT`, `ARG_WAIT` and `RESET_READY`. Therefore, the overhead of `yield` in terms of the number of additional states is constant for every layer. However, to correctly transfer to the next state after the `yield`, the next state is stored in a variable `next_state` and `RESET_READY` transfers on this variable. This makes the state transfer non-deterministic at compile time and prevent some optimization like Xilinx Vivado FSM auto state encoding.

State Transfer of `call` State transfer of `call` is similar to that of `yield` but reorganized, as shown in Figure 2.7. The output valid signal is asserted in the same cycle as the output data is assigned (more closely conforms to the handshaking protocol than that of `yield`). Lowering the ready signal is moved to the beginning of Next State.

2.4 LLVM and Clang

LLVM is a state-of-the-art open-source compiler infrastructure project that provides a comprehensive and modular suite of compiler and toolchain technologies. The LLVM project aims to provide a set of reusable and language-independent software tools that can be used to develop high-performance compilers and dynamic programming language implementations. LLVM's architecture is designed to promote experimentation, research, and innovation in the field of compilers and programming languages, while also offering a high degree of flexibility, portability, and modularity.

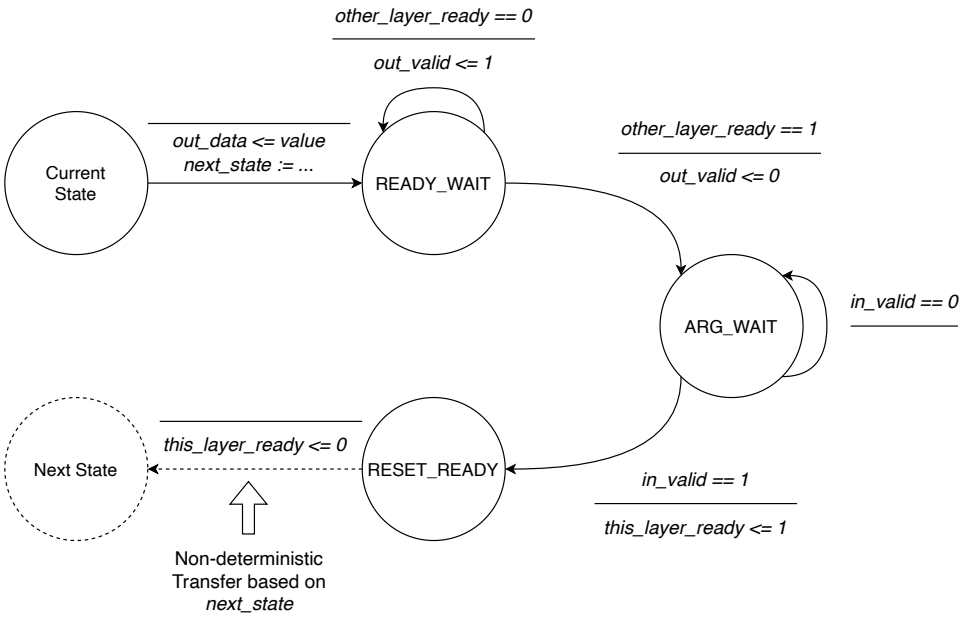


Figure 2.6: State transfer (Mealy) for a `yield` in the previous work [27]. State and signal names are simplified. The dashed line shows a non-deterministic state transfer.

Clang is a C, C++, and Objective-C compiler built on top of LLVM. It is widely adopted by developers and organizations seeking to develop high-performance software systems. It follows the same modular design as LLVM and targets high reusability.

In this project, we build Efeu's compiler based on the LLVM/Clang framework. We intend to reuse components from LLVM/Clang for robustness and extensibility.

2.4.1 LLVM/Clang Pipeline

Inside Clang, the source code undergoes a sequence of stages to be transformed into executable machine code [29]. There is a concise breakdown of the process:

1. **Source Code to Clang AST (Abstract Syntax Tree).** The first step in the Clang pipeline is the conversion of source code into AST. The source code is *preprocessed* to remove preprocessor derivatives. And then, the code is *lexed* and *parsed* into *Clang AST*. The AST captures the rich syntactic and semantic information about the source code, including types, variable names, control flows, and other structures. One can refer back to text positions in the source code from the nodes in the AST.

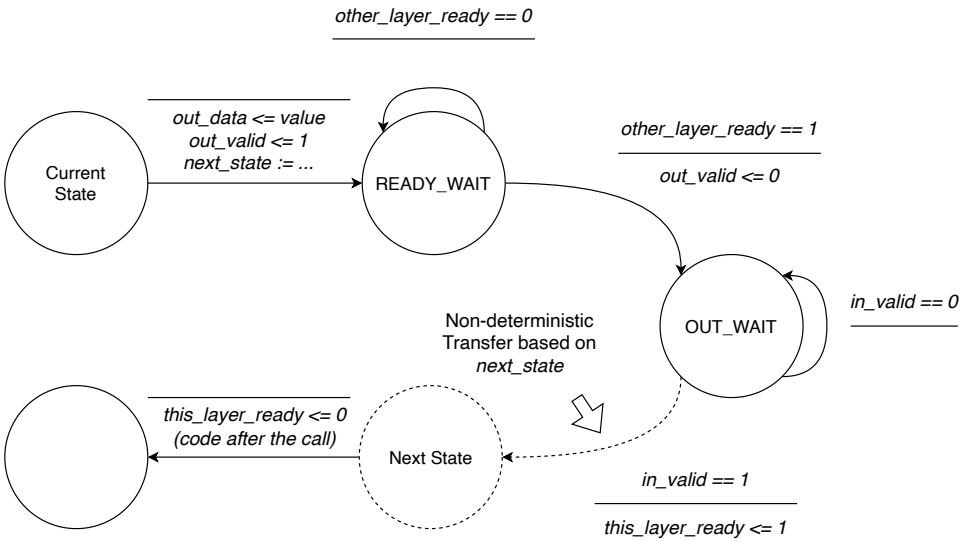


Figure 2.7: State transfer (Mealy) for a `call` in the previous work [27]. State and signal names are simplified. The dashed line shows a non-deterministic state transfer.

2. **Clang AST to LLVM IR (Intermediate Representation).** Once the AST is generated, it is then transformed into *LLVM Intermediate Representation (IR)*. LLVM IR is a low-level, typed, and platform-independent representation. It is designed to be easily manipulated by the LLVM optimization passes. High-level control flows are unified into labels and jumps at this level. Names of variables, functions, etc. can be mangled.
3. **LLVM IR to Assembly.** After the LLVM IR is optionally optimized, it is lowered to target-specific assembly languages. The *LLVM backend*, tailored for the target architecture, is responsible for this conversion.

In addition to the stages described above, Clang also includes various components like linker and runtime supports. We omit them as they are not relevant to the work we present here.

2.4.2 Build System

LLVM/Clang uses the CMake build system. While LLVM-based projects should use CMake, there are still multiple ways to architect LLVM-based projects. In the talk "Architecting out-of-tree LLVM projects using cmake" at the 2021 LLVM Developers Meeting [21], three main architecture options for LLVM-based projects were discussed:

- In-tree source monolithic build: In this architecture, the project's source code is placed within the LLVM source tree and built as part of the

LLVM monolithic build. This structure can be useful for small projects or ones that require deep integration with LLVM, but it can also lead to maintainability issues as changes to the project can impact the larger LLVM codebase.

- Out-of-tree source monolithic build: In this architecture, the project's source code is located outside the LLVM source tree, but is still built as part of a monolithic build system. LLVM source tree is checked out as a Git submodule or with a checkout script. This structure provides better maintainability compared to in-tree builds, but still requires full build of the used LLVM components.
- Out-of-tree source component build: In this architecture, the project's source code is located outside the LLVM source tree and is built as a separate project using CMake. Typically LLVM is located using the `require` derivative in CMake. This structure allows the reuse of pre-built components of LLVM and is therefore the fastest in compile time. However, this architecture imposes limitations on the project. For example, there is no known way to write an out-of-tree LLVM backend[1,2].

In this work, we use the out-of-tree architecture. Rather than introducing passes or components to be reused by many existing LLVM tools, our focus is on constructing tools tailored to our applications. The out-of-tree architecture offers the advantage of leveraging LLVM components as libraries while keeping the codebase lightweight.

2.5 Hardware Description Languages and High-Level Synthesis

Hardware Description Languages (HDLs) are specialized languages used to define and simulate digital circuits and systems. Primary HDLs include VHDL, Verilog and its successor SystemVerilog. These languages are instrumental in designing, simulating, and verifying digital circuits. FPGAs, being reconfigurable hardware, rely heavily on HDLs for their design and functional validation.

HDLs are compiled to bitstreams by Electronic Design Automation (EDA) tools, which is then loaded onto FPGAs. In this work, we use the Xilinx Vivado FPGA tool suite (version 2020.1) as the EDA tool for FPGA implementation and evaluation. The version supports mixed HDLs for design

[1] `https://discourse.llvm.org/t/support-for-out-of-tree-targets/61441`, accessed 2023-04-26.

[2] `https://discourse.llvm.org/t/support-for-out-of-tree-backend-passes/51321`, accessed 2023-04-26.

and simulation. However, only IP blocks with the top-level design written in VHDL or Verilog, but not in SystemVerilog, can be integrated into block designs. This affects some design decisions of ESMC.

EDA tools perform a series of optimizations when synthesizing and implementing the design. For instance, wires and registers that are never used are eliminated. An optimization in Xilinx Vivado that is worth mentioning is the Vivado Finite State Machine (FSM) auto state encoding[3], since FSMs are pervasively used by ESMC when encoding the specification as hardware drivers. When the FSM auto state encoding is enabled and Vivado detects FSM structures in the HDL code, it can choose a suitable state encoding to accommodate the optimization goals. The available encoding methods include one-hot, grey state, Johnson, and sequential. The default optimization goal in Vivado is timing. Depending on specific conditions, Vivado may not detect the FSM or may decide not to perform the optimization. For example, when there exist non-static state transactions, the optimization is disabled[4].

HDLs describe hardware at a low level. In contrast, High-Level Synthesis (HLS) is a process that translates high-level programming languages, such as C, C++, or SystemC, into HDLs. HLS enables hardware designers to describe functionality at a higher level of abstraction, thereby accelerating the design and verification process. Over the years, a number of HLS tools have emerged, ranging from commercial tools, such as Xilinx Vivado HLS, to academic solutions like gcc2verilog [16].

2.5.1 Ready/Valid Handshaking Protocol

The ready/valid handshaking is a flexible and lightweight protocol to connect modules. It is widely adopted in hardware design, such as the AMBI AXI4 protocol [1]. For a unidirectional communication channel, the sender outputs a *valid* signal and data signals, registered by the receiver. The receiver outputs a *ready* signal that is registered by the sender. The two components have to be in the same clock domain. The protocol is defined as follows, assuming signals are asserted on the rising edges.

- In a cycle where valid is high, values on the data ports are valid.
- In a cycle where ready is high, the receiver is able to receive data in that cycle.
- In a cycle where both valid and ready is high, one data transfer is completed.

[3] `https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis/Auto-State-Encoding`, accessed 2023-09-09.

[4] `https://support.xilinx.com/s/article/60104?language=en_US`, accessed 2023-09-09.

Upon a cycle where both valid and ready are asserted high, the sender shall lower the valid signal unless there is an immediate need to transmit more data on the next cycle. Similarly, the receiver shall lower the ready pin unless it can immediately accept more data on the next cycle.

For detailed descriptions and examples of the protocol, readers are kindly directed to the AXI protocol introduction manual [1].

2.6 Enzian and Enzian BMC

The I^2C communication protocol has significant application in the realm of computer platform management. Modern computers are complex. Bootstrapping a system and maintaining its proper operational state involves non-trivial work. Consequently, modern platforms typically incorporate a dedicated processing unit, the *Baseboard Management Controller* (BMC), which handles system initialization, power control, state monitoring, and other basic functionalities [25]. Some BMCs also offer advanced administration capabilities, such as remote control over the network.

While researchers typically have no access to server BMCs, the Enzian project [3] in the Systems Group at ETH Zurich is building an experimental server-class computer for research. Enzian currently includes a BMC with a Xilinx Zynq-7000 System-on-Chip (SoC). The next-generation BMC is an Enclustra Mercury XU5 module [8] that features an ARMv8-based Zynq UltraScale+ MPSoC (ZynqMP) [32]. The Enzian BMC widely uses the I^2C protocol to interact with peripherals, such as the fan controller. Towards the way of building a trustworthy BMC firmware stack, verified I^2C drivers are indispensable parts.

Evaluation of the framework we present in this work is performed on the new BMC module. At the heart of the module is a Zynq UltraScale+ MPSoC (ZynqMP) [32] that features quad-core ARM Cortex-A53 processors as the Application Processing Unit (APU) and a 16nm FPGA (Programming Logic, PL).

The currently deployed Enzian BMCs run OpenBMC [5], a Linux distribution designed for baseboard management. This software stack, although well-tested, does not provide as high assurance as formal verification. The next-generation Enzian BMC firmware is built around seL4 [13, 19], a microkernel that has been formally proven to be functionally correct and secure on specific platforms. Previous work at the Systems Group implements Linux virtualization on seL4 on the ARMv8 platform as a step toward trustworthy BMC firmware. The untrusted components can be put in the VMs for strong isolation, while the trusted parts can be implemented as native seL4 applications for better performance.

Chapter 3

Design

In this chapter, we introduce the design of Efeu. We start by discussing the limitations of the previous works by Humbel [14, 15] and Störzbach [27] in Section 3.1. Section 3.2 presents the design principles of Efeu, which aim to address these challenges and provide enhanced features. This is followed by an overview of the Efeu programming model in Section 3.3 and the new DSLs for Efeu in Section 3.4 and Section 3.5.

3.1 Limitations of the Previous Frameworks

The previous works [14, 15, 27] demonstrate the feasibility of model checking the I^2C stack (including non-compliant devices) and generating software and hardware drivers from the specification. Nevertheless, to generate drivers to be used in real-world applications, there are yet some limitations in these works.

Fixed Bottom-Up Architecture The original I^2C stack specification was designed with a specific direction in mind: the generated C code is intended to be invoked from the lowest layer—the Electrical layer. Figure 3.1 shows a boilerplate code that acts as the Electrical layer and calls into its upper layer. Each of these calls propagates upwards and returns eventually, carrying the SCL and SDA levels to be set on the next cycle back to the Electrical layer.

Consider a simple program operating the I^2C bus. Given the current architecture, this program must spawn a different thread to run an infinite loop akin to the one in Figure 3.1. The thread needs to synchronize with the main thread for I^2C operations. This approach deviates from the conventional method of writing or using drivers. Software drivers usually adopt a top-down approach: they receive high-level instructions, execute operations, and return the results.

```
1  void controller_Symbol(int scl, int sda, int *out_scl, int
       *out_sda);
2
3  int main() {
4    /* some more code */
5
6    int scl, sda;
7    while(1) {
8      i2c_recv(&scl, &sda);
9      controller_Symbol(scl, sda, &scl, &sda);
10     i2c_send(scl, sda);
11   }
12 }
```

Figure 3.1: Original boilerplate code that interacts with bit-banging I^2C driver. This main function acts as the Electrical layer. It calls into its upper layer, the Symbol layer.

Dead-End Top Layer In the original specification, the top layer of the controller stack—the World layer, continuously executes a predetermined sequence of operations, as shown in Figure 3.2.

```
1  proc (int /* action */, int /* addr */, int /* data_len */,
       intarr /* data */) EepController(int res, intarr res_data) {
2    intarr data;
3    assert(res == ME_RES_IDLE);
4
5  write:
6    data[0] = 32;
7    data[1] = 33;
8    yield (ME_ACT_WRITE_EEPROM, 6, 2, data);
9    assert(res == ME_RES_OK);
10
11 read:
12   yield (ME_ACT_READ_EEPROM, 6, 2, data);
13   assert(res == ME_RES_OK);
14   assert(res_data[0] == 32);
15   assert(res_data[1] == 33);
16
17   goto write;
18 }
```

Figure 3.2: Original top layer of controller that executes a predetermined sequence of operations indefinitely. The `proc` is named differently in this piece of code but corresponds to the World layer describes in [15] and [14].

While the generated drivers work for testing purpose, real applications require drivers that can accept inputs from outside. Although it is possible to use methods like `scanf` in the World layer, or acquire data through other approaches, a more refined solution is preferable.

DSL Usability From the usability standpoint, the original DSL used to specify the stack has several limitations. The DSL is similar to C, yet it is neither a subset nor a superset. Unlike C, which has a complete ecosystem of tools like code highlighting, formatting, and compilers with comprehensive diagnosis, the DSL is only equipped with a basic compiler. In addition, as highlighted in Subsection 2.3.2, the usage of `call` is unclear from the definition of the current layer but instead determined by the upper layer's signature. The `yield` usages can also be perplexing, as those "function parameters" that appear to be static can change after the call.

Drivers are Solely Software or Hardware Humbel's compiler [14, 15] compiles the whole stack into C code for software drivers. Störzbach's extension [27] compiles the entire stack into VHDL code for hardware drivers. There is no way to generate software-hardware combined stacks.

3.2 Design Principles of Efeu

In this work, we propose an approach to migrate the issues in the previous works [14, 15, 27] mentioned above. At a high level, the core idea is to unify `call` and `yield` to a generic `talk` function, and let the compiler decide the implementation. This change enables a range of new opportunities.

Generating Both Bottom-Up and Top-Down Software Drivers In Efeu, specifications are no longer limited to the bottom-up architecture. By delegating the decision of actual implementations of the `talk` function calls to the compiler, the user can generate both top-down and bottom-up drivers from the same stack, merely by changing a command-line option.

Open Interfaces at Endpoints Unlike the previous works that have a dead-end top layer, drivers generated by Efeu have open and well-defined interfaces at endpoints. For instance, in a top-down software driver generated by Efeu, the top level is a callable C function. The entire stack can be seamlessly encapsulated into a library for programs to invoke.

Creating Complex Layer Topologies Layers connecting to each other form a graph. In the previous framework, using the `yield` and `call` operations, the user can only construct linked-list-like structures, in which each layer is connected to two adjacent layers (except the two endpoints). While this suffices for I^2C drivers, Efeu offers greater flexibility in layer assembly, enabling the creation of more complex topologies. Efeu allows creating a *network* of layers.

Generating Combined Software-Hardware Drivers with One Click Efeu empowers users to produce integrated software-hardware drivers. The split point between software and hardware components can be configured with command-line options passed to the compiler. Moreover, Efeu supports creating drivers that cross the software-hardware boundary multiple times.

The DSL and the compiler are also completely redesigned, with the following goals in mind.

Usability for End Users The new DSL is a strict *subset of C*, which means existing tools like code highlighters and formatters work for the DSL seamlessly. Leveraging the existing frontend of Clang/LLVM, the new compiler can perform type checking as strong as C and report errors and/or warnings in a user-friendly way. Most checks for code style in Clang are retained. The syntax of the `talk` function is designed to avoid confusion. Furthermore, the new DSL supports C preprocessor derivatives, enabling flexible and modular designs.

Ease of Integration with Boilerplate Code or Existing Hardware Design The new compiler can translate system specifications into C or HDL code without user intervention. However, in many real-world applications, the generated code needs to be integrated with boilerplate code (for software drivers) or pre-existing hardware design (for hardware drivers). The DSL and the compiler are designed with this integration in mind. For software drivers, the end user can even incorporate native C code (out of the DSL)'s scope) directly within the specification. For hardware drivers, the generated code can be easily encapsulated into IP blocks in EDA tools with little manual effort.

Last but not least, Efeu retains the model checking capability of the original frameworks. The compiler is able to generate Promela code for the SPIN model checker. Efeu effectively encompasses and extends the capabilities of the earlier work.

3.3 Programming Model

Layers (Nodes) Efeu inherits the concept of *layers* from the previous works. Layers are processes (coroutines) running indefinitely. At its core, a layer operates as a Finite State Machine (FSM). A single layer can be connected to one or more layers. In this sense, layers are better named as *nodes*. However, for the sake of consistency, we still use the word "layers" in the remaining part of this work. Any pair of layers can be connected or disconnected, but not connected multiple times.

Interface An *interface* is a *unidirectional* communication channel from one layer to another. Connections between layers are always *bidirectional*. As a result, two connected layers `A` and `B` have *two* interfaces between them, one from `A` to `B`, and the other from `B` to `A`. An interface contains zero or more *data fields*, with each data field processing a specific type. Available types in Efeu include:

- Bit: 0 or 1.
- Boolean: true or false.
- Byte: unsigned 8-bit integer.
- Signed 16-bit integer.
- Signed 32-bit integer.
- A fixed-size array of the above type.

All types have fixed sizes that can be determined at the compile time. Accordingly, interface sizes, as the sum of their corresponding data fields, are also predetermined.

Network A *network*, or layer topology, is a *undirected* graph composed of layers and interfaces between them. Given connections between layers are always bidirectional, the network is undirected. On every edge of a network, there are two interfaces (directed).

The original linked-list-like structure qualifies as a network: each layer is connected to two other layers, with the exceptions being the two endpoints that connect to just one layer each. The concepts of the top/bottom layer and upper/low layer are undefined unless a direction is defined (for example, based on the level of abstractions in the I^2C communication stack). For consistency with the previous work, in the remaining part of this work, we might still use the terms "upper" and "lower" layers. Readers shall differentiate if this direction is defined by the I^2C abstraction levels or by a direction assigned to the layer topology.

Component A *component* refers to a subgraph of the network that is processed as a unit through one backend in the Efeu compiler. The whole network can be a single component. For instance, processing the whole network as one component through the C backend results in a pure-software driver codebase. Conversely, a network split into a C component and an HDL component results in a combined software-hardware driver.

The `talk` Operation The `talk` operation is the successor of `yield` and `call` in the previous works. Consider two connected layers `A` and `B`. Each

(directional) interface has an associated with a `talk` operation. With the two interfaces denoted as `AToB` and `BToA`, the corresponding `talk` functions are `ATalkB` and `BTalkA`. A layer can only `talk` on the interfaces that originate from the current layer, thereby `ATalkB` is only invokable in `A`, and vice versa. `ATalkB` has the following semantics, executed in sequence:

1. Transmit values from `A` to `B`. Every data field in the outgoing interface `AToB` must have a value associated with it.
2. Wait until `B talks` back via `BTalkA`.
3. Retrieve the values from `B` through its `talk` operation. It is guaranteed that every data field in the incoming interface `BToA` has a value associated with it.
4. The execution resumes right after the `ATalkB` call.

Note that `talk` is a blocking operation. It only returns when the other side `talks` back.

The `read` Operation The `read` operation has the same semantics as `talk` except that it does not transmit values to the other layer (skipping step 1). The equivalence of this operation in the original DSL is implicit—each process gets a set of values from the lower layer on entry. Correspondingly, each layer written in the new DSL always has a `read` operation at the entry of the state machine, but is only executed once.

In the following sections, we present the DSLs of Efeu.

3.4 ESM: Efeu State Machine

The main parts of the specification for an Efeu-based system are the finite state machines of layers. The DSL is named ESM, short for Efeu State Machine. Syntactically, ESM is a subset of the C programming language. We make this design decision for usability considerations. Tools designed for C are directly applicable to ESM, such as code highlighting, code formatting, static checking, and IDEs (Integrated Development Environment). From the implementation perspective, the Clang frontend can process ESM as C without modifications, while still offering users type checking and sophisticated diagnosis.

Subset of C ESM is syntactically a subset of C. Some features of C are not part of ESM and some syntactical constructions have modified semantics.

- Allowed types: `bit`, `bool`, `byte` (`unsigned char`, unsigned 8-bit integer), `short` (signed 16-bit integer), and `int` (signed 32-bit integer). `bit` and `bool` are `typedefed` as `unsigned char` syntactically but semantically

represents one-bit integers. Other built-in types like `float` and `double` are not supported.

- Pointers are not supported.
- No global variables.
- Supported unary operations: plus (`+`), minus (negate, `-`), not (`~`), logical not (`!`). Other operations like address (`&`), dereference (`*`), and pre- or post-increment (`++`), are not allowed.
- Unsupported binary operations: comma (`,`).
- Supported control flow statements: if, while, goto. Other statements such as do-while and switch are not supported.
- No `struct` definitions except those for interfaces (described below).
- No other function declarations or definitions except those for layers and `talk`/`read` functions (described below). Layer functions encode processes running indefinitely. No `return`.
- No initial values on variable declarations.
- Additional reserved keyword: Promela built-in keywords [6], such as `len` and `timeout`.

The language design is affected by several factors, including Efeu's necessary functionalities, constraints from backend languages, and the complexity of implementation. Future improvements are discussed in Section 7.3.

Interfaces Defined as Structs Interfaces are defined using the `typedef ... struct` format. Each data field in the interface translates to a field in the struct. Semantically, there is no padding between fields, and adjusting the padding option is not supported. Lines 7–11 and 13–16 in Figure 3.3 show two examples.

Array types are handled in a more complex way. For each element type and array size, a struct wrapper is generated. For example, if an interface contains a field of an `int` array with size 16, a struct named `intArray16` is created (as seen in Line 5 in Figure 3.3). If another array field shares the same type and size, the struct is reused. To access the array, users need to use `.e.x[i]` instead of `.e[I]`.

The main motivation behind this design choice is that Promela does not allow passing raw arrays through channels. By wrapping the array in a struct, it can then be passed. The previous works [14, 15] employ similar array wrappers for array fields, although the 16-slot int array is the only supported type. Staying consistent with them also helps minimize the adaptions required for the existing verifiers to function.

```
1  #ifndef ESM_EXAMPLE_ESI
2  #define ESM_EXAMPLE_ESI
3
4  typedef struct { byte x[4]; } byteArray4;
5  typedef struct { int x[16]; } intArray16;
6
7  typedef struct {
8    bit a;
9    bool b;
10   byteArray4 c;
11 } FooToBar;
12
13 typedef struct {
14   short d;
15   intArray16 e;
16 } BarToFoo;
17
18 #define PREAMBLE_Foo  \
19   extern BarToFoo FooTalkBar(bit a, bool b, byteArray4 c);  \
20   extern BarToFoo FooReadBar();
21
22 #define PREAMBLE_Bar  \
23   extern FooToBar BarTalkFoo(short d, intArray16 e);  \
24   extern FooToBar BarReadFoo();
25
26 #endif
```

Figure 3.3: An example of interface and function declarations to be included in ESM files. It is also the header generated from ESI in Figure 3.6. The header guard is generated based on the filename.

Declarations of `talk` and `read` Functions For each layer, available `talk` and `read` functions are declared as extern C functions within a macro definition. The macro is expected to be used at the beginning of the corresponding layer function body. This approach, compared with declaring the functions in the global space, helps limit the scope of these functions. While the ESM compiler asserts the correct usages of these functions, possible misuses in wrong layers can be prevented early in development with static checkers and IDEs. Output parameters to the other layers are passed as individual parameters. The values are passed back as a struct. Lines 18–20 and 22–24 in Figure 3.3 show two examples.

Layers as Functions Layers are defined as void-returning functions with no parameters. Layer functions encode processes running indefinitely and thereby should never return. At the beginning of the function, the preamble corresponding to the layer should be included to enable `talk` and `read` functions in the function body, as shown at Line 4 in Figure 3.4.

```
1  #include "example.esi.h"
2
3  void Foo() {
4      PREAMBLE_Foo
5  #define talkBar FooTalkBar
6  #define readBar FooReadBar
7
8      BarToFoo b;
9      b = readBar();
10
11 loop:
12     /* ... */
13
14     b = talkBar(0, true, 42);
15
16     /* ... */
17
18     goto loop;
19 #undef talkBar
20 #under readBar
21 }
```

Figure 3.4: An example of ESM files. The header being included is shown in Figure 3.3.

Preprocessor Derivatives Preprocessor derivatives *are supported* in ESM. For instance, at Line 5 and 6 in Figure 3.4, shorter aliases are set for the `talk` and `read` functions. In addition to aliasing functions, users also can implement polymorphism using macros. In the previous DSL [14, 15], processes support compile time arguments. In ESM, this functionality is achieved by macros. Later in this work, Figure 4.15 shows another example of reusing the same state machine in multiple layers using `#includes`.

3.5 ESI: Efeu System Information

Interface definitions influence both the struct definitions and the `talk/read` declarations. Manually crafting the template code like Figure 3.3 is error-prone. Moreover, there are several constraints that need to be enforced on the interface definitions. For instance, every pair of connected layers must have two and only two interfaces in between. With end-user usability in mind, we design another lightweight, declarative DSL for defining layers and interfaces. The DSL is termed ESI, short for Efeu System Information. With ESI, users are encouraged to design the system in a top-down approach: first define the interfaces between layers, and then design the state machines.

Figure 3.5 shows the syntax of ESI. The language specifies layers and interfaces between them. Each pair of connected layers should have exactly two interfaces. The directions of interfaces are annotated with the keywords `=>`

and <=. In each interface, data fields are defined with a type and a name. Single-dimensional arrays of elementary types are supported. Figure 3.6 shows an example of two layers with the interfaces between them. Later in this work, Appendix A shows the complete ESI of the I^2C stack.

$$
\begin{aligned}
\langle \text{file} \rangle &\models (\langle \text{layerDecl} \rangle \mid \langle \text{interfaceDecl} \rangle) * \texttt{EOF} \\
\langle \text{layerDecl} \rangle &\models \texttt{layer}\ \langle \text{identifier} \rangle\ \texttt{;} \\
\langle \text{interfaceDecl} \rangle &\models \texttt{interface <}\langle \text{identifier} \rangle\texttt{,}\langle \text{identifier} \rangle\texttt{> \{} \\
&\quad \langle \text{direction} \rangle \langle \text{structDecl} \rangle\texttt{,}\langle \text{direction} \rangle \langle \text{structDecl} \rangle\texttt{,?} \\
&\quad \texttt{\}} \\
\langle \text{direction} \rangle &\models \texttt{<=} \mid \texttt{=>} \\
\langle \text{structDecl} \rangle &\models \texttt{\{} (\langle \text{type} \rangle \langle \text{identifier} \rangle) * \texttt{\}} \\
\langle \text{type} \rangle &\models \texttt{bit} \mid \texttt{bool} \mid \texttt{u8} \mid \texttt{i16} \mid \texttt{i32} \\
\langle \text{identifier} \rangle &\models \langle \text{char} \rangle\, [\langle \text{char} \rangle \mid \langle \text{digit} \rangle] * \\
\langle \text{char} \rangle &\models \texttt{A...Z} \mid \texttt{a...z} \mid \texttt{\_} \\
\langle \text{digit} \rangle &\models \texttt{0...9}
\end{aligned}
$$

Figure 3.5: Syntax of ESI. Entities are separated by one or more spaces, tabs, and/or new lines, which are omitted for brevity.

```
1  layer Foo;
2  layer Bar;
3
4  interface <Foo, Bar> {
5    => {
6      bit a;
7      bool b;
8      u8 c[4];
9    },
10   <= {
11     i16 d;
12     i32 e[16];
13   },
14 };
```

Figure 3.6: An example of ESI. => annotates the interface from `Foo` to `Bar`. <= annotates the reversed interface. This piece of code translates to Figure 3.3

ESI does not directly integrate with ESM code. Instead, users generate an ESM header from ESI and use that header in ESM files. However, thanks to the conciseness of ESI, the ESI file is still supplied to the compiler as a

trusted specification when compiling the ESI files. If an interface struct or a function signature in the ESM code is inconsistent with the ESI, ESMC trusts the ESI and attributes the inconsistency to the user's ESM code.

Chapter 4

ESMC: ESM Compiler

After users specify the system topology with ESI and state machines with ESM, the ESM compiler, ESMC, compiles them to various outputs based on the user's instruction: Promela code for model checking, C code for software drivers, and/or Verilog/VHDL code for hardware drivers. Generating ESM headers from ESI is also performed by ESMC. All these functionalities are integrated into a single program. Users specify the desired output with command-line options.

Figure 4.1 illustrates the user workflow using ESMC. As discussed in Chapter 3, the first step of designing a system with Efeu is writing the ESI file and generating a header to be included in ESM files, denoted as ① in Figure 4.1. After that, the user writes ESM code using the generating header. The second phase, denoted as ②, involves supplying the ESI and ESM files to ESMC, which then produces the desired user output.

ESMC also supports the whole system compilation. Given a system configuration, ESMC automatically steers the corresponding backend pipelines to generate a composition of software and/or hardware components. We discuss this functionality in Section 4.8.

Internally, ESMC leverages several components from Clang/LLVM, including the Clang Lexer, Parser and Codegen. However, it does not follow the LLVM frontend-backend design mode but develops its own dataflow. Figure 4.2 shows the internal dataflow of ESMC with Promela backend flow in orange, C backend flow in blue, and HDL backend flow in green. Notably, the HDL flow goes through the Clang Lexer and Parser twice, denoted with ① and ②. For clarity, the dataflow of ESI is not shown. ESI code is lexed, parsed, and compiled by ESMC and is used in all backends.

In the following sections, we describe the implementation details of components in ESMC.

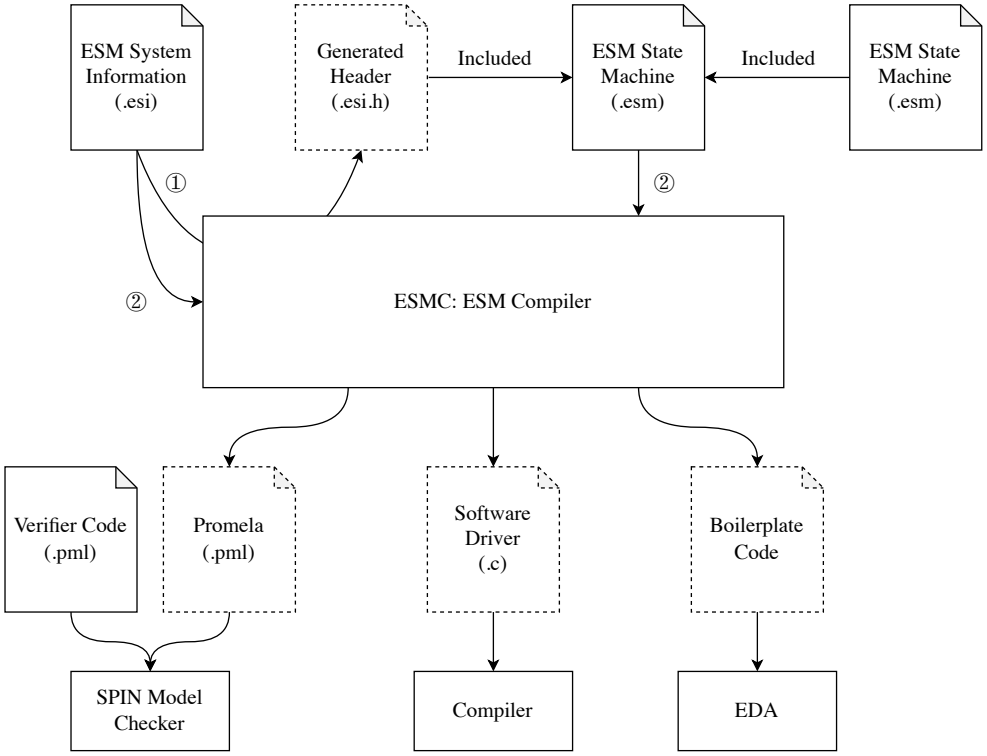


Figure 4.1: ESMC workflow. Files with solid borders are written by the user and the dashed ones are generated.

4.1 ESI Frontend

To lex and parse ESI, we develop a lexer and a parser. We define ESI AST in LLVM style[1]. ESI lexer generates a token stream from the input ESI file. Subsequently, the ESI parser takes the token stream and constructs an AST. From the AST, an ESI compiler performs various checks. For example, it ensures that there is at most one (or none) pair of interfaces defined between any two layers. If no problem is detected, the internal presentation of ESI is generated to be used in other components. Otherwise, errors are reported to the user.

4.2 ESI to ESM Header Backend

The ESM header backend processes the internal representation of ESI and emits the ESM header according to the rules discussed in Section 3.4 and Section 3.5. The translation is straightforward. The header guard is generated from the input filename by capitalizing all letters and substituting

[1] `https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html`

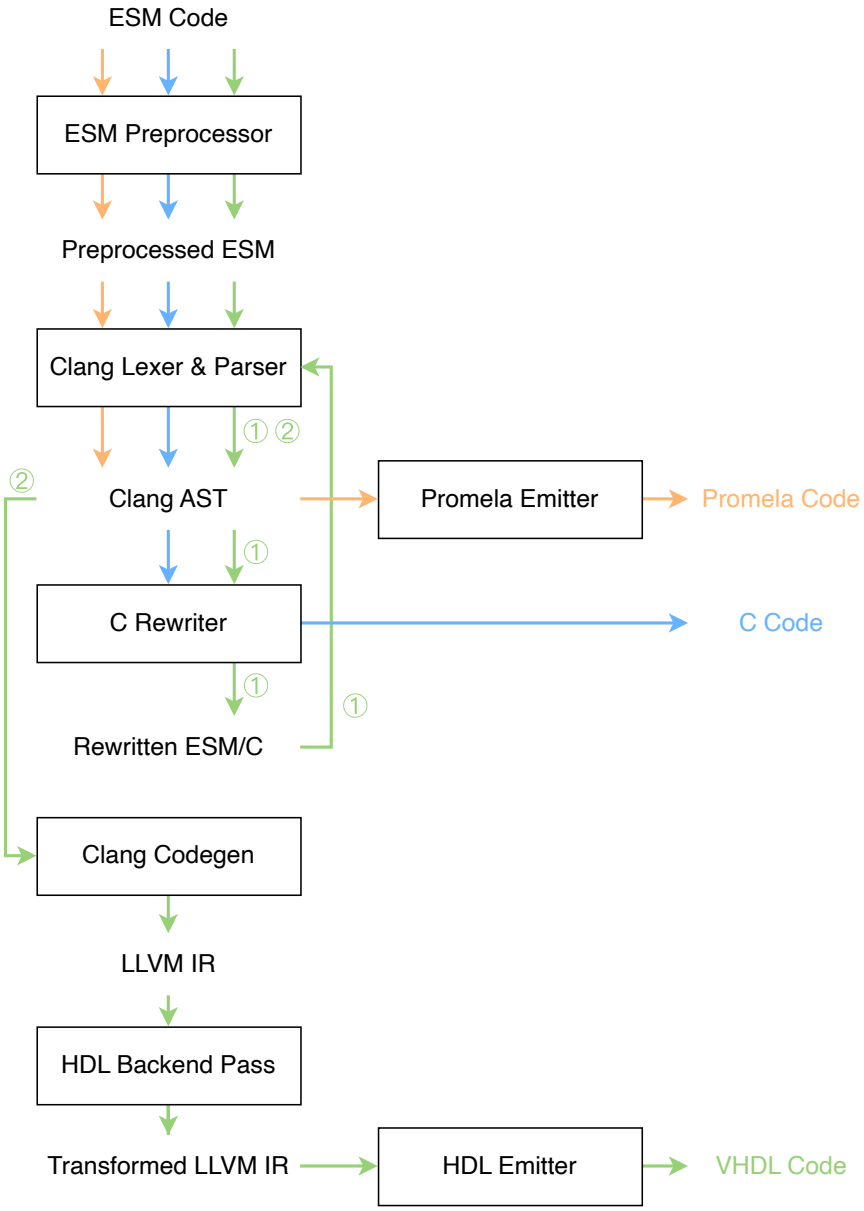


Figure 4.2: ESMC dataflow. Orange: Promela backend flow. Blue: C backend flow. Green: HDL backend flow.

any non-alphanumeric characters with underscores, then prefixed with `ESM_`. For example, if the input filename is `i2c.esi`, the header guard macro is `ESM_I2C_ESI`.

4.3 ESM Frontend

As discussed in Section 3.4, the ESM language is a subset of C. This allows us to reuse the existing Clang frontend to lex and parse ESM files. In addition, static type checking and built-in diagnosis (including code style warnings) are immediately available. If there is no errors during parsing, the frontend generates a Clang AST, which is subsequently directed to backends (discussed in the following sections).

From the implementation perspective, ESMC adopts a similar construction as Clang. `clang::CompilerInstance` is the top-level object that manages the whole complication process. The input ESM file is supplied as a C file. Upon invoking `CI.ExecuteAction`, each input file is preprocessed, lexed into a token stream, parsed into a Clang AST, and fed to the `ASTConsumer` specified in the `FrontendAction`. Diagnoses are managed by `clang::DiagnosticsEngine`, to which Clang frontend outputs warnings, errors, and notes. Our `FrontendActions` can also raise customized warnings and/or errors.

4.4 Promela Backend

As mentioned in Section 2.4, Clang adopts multiple intermediate representations through its compilation pipeline. For ESMC backends, there are decisions to be made regarding which representations to use as the starting points. One option is to start from the Clang AST. The source code undergoes the preprocessor, the lexer, and the parser to reach this stage. The AST preserves the source-level information with all macro expanded and type checked. Another option is the LLVM IR. From the Clang AST, it additionally goes through the IR generator and possibly optimization passes. At this stage, control flow structures are flattened to labels and jumps. Information like variable names is wiped out.

The Promela backend starts with the Clang AST. The primary motivation is that the Clang AST retains variable names, while LLVM IR does not. After ESM is translated to Promela, it still needs to be combined with external Promela code (including the code that generates valid inputs and the code that asserts outputs) to create the complete verifier. In this case, variable names must be preserved so that those Promela code pieces match.

Furthermore, the Clang AST preserves source-level control flow structures. When the SPIN model checker reports issues, the user needs to map the Promela code back to the ESM code to debug. By starting from the Clang AST, the mapping is mostly intuitive. In contrast, if we start with LLVM IR, reverse engineering the control flow is cumbersome, if not impossible.

For these reasons, we implement the Promela backend to take the Clang AST as the input and the Promela code as the output. In the codebase, the main class is `PromelaEmitter`. It acts as a `clang::RecursiveASTVisitor` that traverses the Clang AST and emits Promela code in text.

Promela syntax closely resembles C syntax. A large portion of the syntax can be translated directly to Promela, such as variable declarations, operators, `goto` statements, etc. We discuss the notable parts of the translation rules in the following subsections. The code emitted by the Promela backend is properly indented.

4.4.1 Types

All ESM built-in types (`bit`, `bool`, `byte`, `short`, and `int`) are translated one-to-one to Promela types of the same names. As discussed in Section 3.4, `bit` and `bool` are not built-in types of C but `typedefs` (as `unsigned char`) through `esm.h`. However, Clang AST preserves the typedef information, and therefore `PromelaEmitter` can exactly map them to `bit` and `bool` in Promela. Definitions of structs for interfaces and array wrappers are translated into `typedefs` in Promela.

4.4.2 Enums

Promela supports `mtype` [6] for enumerations. It is essentially equivalent to a global enum but with descending numbering down to 1. To faithfully retain the C semantics of enums, ESMC converts them into macro definitions with proper values assigned, and enum variables into `int` variables. Examples can be seen in Figure 4.3).

ESM

```
typedef enum {
  SYM_IDLE,
  SYM_START,
  SYM_STOP,
  SYM_BIT0 = 9,
  SYM_BIT1,
  SYM_STRETCH,
} Symbol;

typedef struct {
  Symbol sym;
} FooToBar;
```

Promela

```
#define SYM_IDLE 0
#define SYM_START 1
#define SYM_STOP 2
#define SYM_BIT0 9
#define SYM_BIT1 10
#define SYM_STRETCH 11

typedef FooToBar {
  int /* Symbol */ sym;
};
```

Figure 4.3: ESM enum to Promela.

4.4.3 ESM Interfaces to Promela Channels

Each interface from one layer to another is encoded as a rendezvous port (or synchronous) channel (with buffer size 0) in Promela [6]. Sending to or receiving from the channel is blocking, until the other side receives from or sends to the channel (hence the term "rendezvous"). Such a channel encodes the same semantics as the handshaking protocol: a transaction is done when both sides are ready. Figure 4.4 shows an example piece of Promela code containing channels between layers.

```
1  typedef byteArray4 {
2    byte x[4];
3  };
4  typedef intArray16 {
5    int x[16];
6  };
7
8  chan FooToBarChan = [0] of { bit, bool, byteArray4 };
9  chan BarToFooChan = [0] of { short, intArray16 };
10
11 typedef FooToBar {
12   bit a;
13   bool b;
14   byteArray4 c;
15 };
16
17 typedef BarToFoo {
18   short d;
19   intArray16 e;
20 };
```

Figure 4.4: Promela generated by only including the sample header (Figure 3.3) in the ESM code, which is generated from the sample ESI (Figure 3.6).

4.4.4 ESM Functions to Promela Processes

As discussed in Section 3.4, a layer function in ESM is equivalent to an ever-running process, which exactly resembles a Promela process. Therefore, each layer function is translated to a corresponding Promela process.

One thing to note is that while ESM (its underlying C syntax) supports multiple declarations (but not definitions) of the same function, Promela does not support that. As a consequence, additional function declarations must be eliminated. Figure 4.5 shows an example of such encoding.

ESM

```
void Foo();

void Foo() {
  int a;
  a = 42;
}
```

Promela

```
proctype Foo() {
  int a;
  a = 42;
}
```

Figure 4.5: Encoding of an ESM layer function to Promela process.

4.4.5 Control Flows

Promela provides non-deterministic control flow constructs (Section 2.2) that have different semantics from C. They need to be handled with care. Given the allowed subset of C syntax in ESM, the relevant Promela control flow constructs are `if` [6] and `do` [6].

Figure 4.6 shows an example encoding of the if statement. In C, `if-else if` are evaluated in order. Therefore, when translated to Promela, an `else if` should be translated to an `if` nested in the `else`, rather than another non-deterministic choice of the previous `if`. Also, notice that an `else -> skip` block is generated even if the `if` statement has no `else`. This is necessary. In Figure 4.6, consider the case where `a != 0`. In C semantics, the execution continues after skipping the whole body. In Promela semantics, if the `else -> skip` is missing, the `if` statement becomes blocking until the condition becomes true.

ESM

```
if (a != 0) {
  if (a == 1) {
    b = 1;
  } else if (a == 2) {
    b = 2;
  } else {
    b = 3;
  }
}
```

Promela

```
if
:: (a != 0) ->
  if
  :: (a == 1) ->
    b = 1;
  :: else ->
    if
    :: (a == 2) ->
      b = 2;
    :: else ->
      b = 3;
    fi
  fi
:: else -> skip;
fi
```

Figure 4.6: Encoding of the if statement from ESM to Promela.

Similarly, when encoding `while` in C as `do` in Promela, an `else -> break` case is a must. Otherwise, the loop blocks when the condition is evaluated to false, which owns a different semantics. Figure 4.7 shows an example of the loop encoding.

ESM

```
while (a < 10) {
  while (b < 20) b += 2;
  a += 1;
}
```

Promela

```
do
:: (a < 10) ->
  do
  :: (b < 20) -> b = b + 2;
  :: else -> break;
  od
  a = a + 1;
:: else -> break;
od
```

Figure 4.7: Encoding of the while statement from ESM to Promela.

4.5 C Backend

The ESMC C backend transforms ESM code into C code, which can then be compiled into executables or libraries as software drivers. The backend is an extension of the compiler in previous work [14, 15], supporting the new ESM syntax and the generalized `talk` and `read` functions (Section 3.4).

While ESMC is built on the Clang/LLVM framework that contains the complete complication pipeline, the C backend emits C source code rather than pushing it all the way to binary code. In this sense, the backend is more of a rewriter than a compiler. The primary rationale behind this approach is to ensure portability. The target platform for the generated stack may be in a different architecture (for example, the new BMC is ARMv8-based). Cross-compiling or compiling on the target machine is sometimes necessary. Additionally, the target platform may run a different OS such as seL4 [19]. The generated code is platform-independent, offering flexibility to be combined with OS-specific boilerplate code.

Efeu layers have the semantics of ever-running processes. A straightforward way to implement it is to encode layers as threads and `talk`/`read` as inter-thread communications. However, this approach introduces scheduling overhead. In addition, the code becomes less portable by using OS-specific thread implementations. Instead, we focus on a static scheduling strategy proposed by the previous work [15].

For software like I^2C drivers, concurrency in layers is not a required feature for functionalities. While layers are modeled as processes, the whole stack converges to *a single execution point* in one layer after initialization, while other layers wait for replies from others. Whenever a layer `yield` or `call` (`talk` in ESM), the layer hands out the execution handle and starts waiting, while the target layer becomes active. Thereby, at any point, *only one* layer is executing, while others are guaranteed to be waiting.

Systems with such property can be implemented as a single-threaded program with C function calls and returns, without any concurrency abstractions provided by the OS. Note that the semantics between the specification and

the generated C code have not been formally proved. We briefly discussed it later in Section 7.3 and it remains as a future work. But if limiting our application to the I^2C stack, the strategy has been proven functional for the stack on real systems.

There are three main components in the C backend: the calling tree solver, which decides how `talk`and/or `read` functions should be transformed; a customized preprocessor, which processes macros like `#include` to prepare the code for rewriting; and the core component—a rewriter, which transforms ESM code to C code. We detail these three components in the following subsections. For clarity, we discuss the rewriter before the customized preprocessor, although they are executed in the reversed order in ESMC.

4.5.1 Calling Tree and Solver

The previous framework transforms `call`s into C function calls and `yield`s into continuations. In Efeu, we found the equivalence between `call` and `yield`, thereby unifying them into `talk`. When generating C code, the actual implementations, either a C function call or a continuation, are decided by the *calling tree*.

The calling tree is a tree subgraph of the layer network. Given a *calling point* as one of the layers, the calling tree is retrieved by performing a depth-first search (DFS) on the layer network starting from the calling point. Directed edges become C function calls, while the reversed edges become continuations.

Figure 4.8 shows examples of calling trees. The two calling trees on the left only use the I^2C controller stack but employ different layers as the calling point. The bold solid arrows represent edges of the calling tree, which are implemented as direct function calls. The reversed edges, annotated in dashed arrows, are implemented as continuations. By simply changing the calling point, without modifying the stack, the user can generate both top-down and bottom-up drivers. On the right side, Figure 4.8 shows a calling tree by assembling a controller and a responder with a modified Electrical layer `talk`ing to both devices in turn. Combined with proper boilerplate code, an I^2C simulator can be constructed.

In calling trees, one layer can call multiple layers, but every reachable layer has a unique caller. This property ensures that only one interface pair is mapped as the parameters of the resulting function. When a layer is called, all input passed-by-value parameters have valid values and all output pass-by-reference parameters are non-NULL.

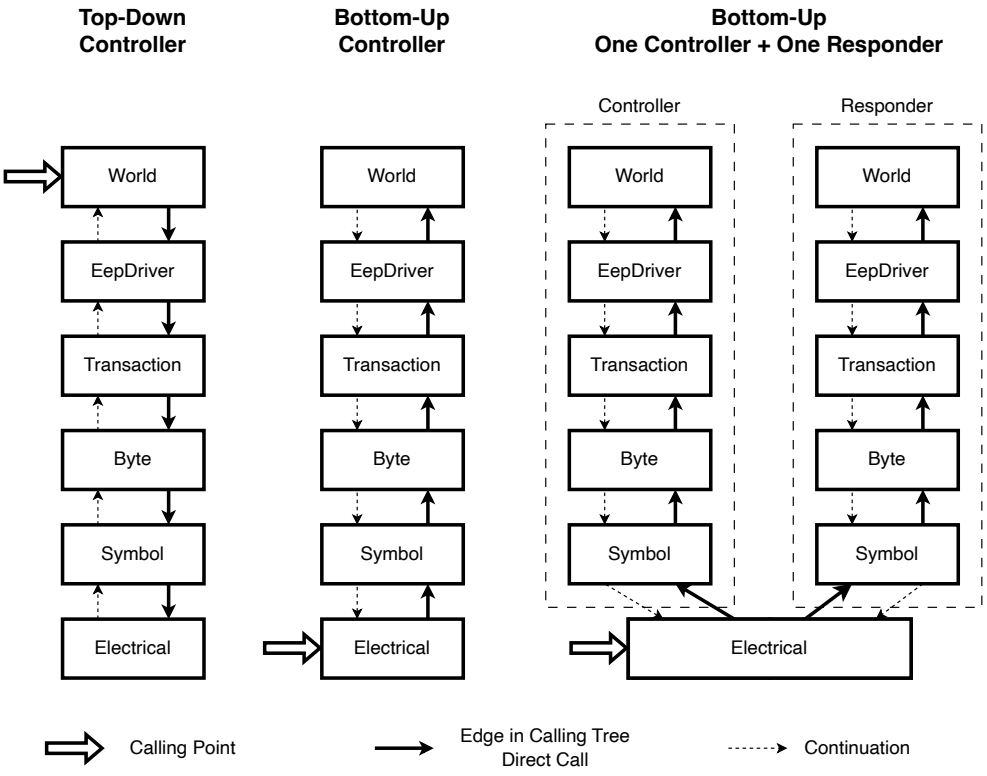


Figure 4.8: Examples of the calling tree.

4.5.2 Rewriter

The C backend takes ESM as input and outputs C. Essentially, it is a rewriter. There are three rewriting processes that need to be performed. In the ESMC codebase, they correspond to three classes: `VarDeclRewriter`, `TalkAndReadRewriter`, and `LayerFunctionRewriter`.

The rewriters are based on the Clang AST Matcher and Rewriter. The Clang AST Matcher is a DSL that describes the structure to be matched in the Clang AST. Thanks to C++ templates, the target structure can be described directly in C++ code. Matched elements are retrieved with customized names. Examples follow as we discuss the rewriting steps below. The Clang Rewriter provides APIs to rewrite source code. It works on source text rather than the Clang AST. The most important APIs are inserting, replacing, or deleting a range of text. Those ranges are provided by the Clang AST. Notice that Clang Rewriter keeps changes in a buffer and the Clang AST remains unchanged after each step. Therefore, it is important that the three steps stay independent and do not interfere with each other.

Firstly, as discussed in the Section 3.4, local variables need to be converted to

static global variables that preserve the values across function calls. This step is performed by the `VarDeclRewriter`. Figure 4.9 shows the corresponding AST matcher. It matches a function (denoted as `func`) that has a function body (not only a declaration). For each variable declaration inside (denoted as `decl`), the AST matcher is triggered once.

```
functionDecl(
  hasBody(
    stmt(
      forEachDescendant(
        declStmt(has(varDecl())).bind("decl")
      )
    )
  )
).bind("func");
```

Figure 4.9: Clang AST matcher to match variable declarations in functions.

Figure 4.10 shows the simplified code of the rewriting process. It first asserts that the enclosing function is a known layer function, otherwise it is skipped. And for each variable declaration that is not static already, keyword `static` is inserted before.

```
virtual void run(const MatchFinder::MatchResult &Result)
    override {
  const FunctionDecl *Func =
      Result.Nodes.getNodeAs<clang::FunctionDecl>("func");
  assert(Func != nullptr);

  // Only handle layers, but allow other functions
  if (!ESI->isLayer(Func->getName()))
    return;

  const DeclStmt *Decl =
      Result.Nodes.getNodeAs<clang::DeclStmt>("decl");
  VarDecl *Var = dyn_cast<VarDecl>(*Decl->decl_begin());
  assert(Var != nullptr && "unhandled case other than VarDecl");
  if (!Var->isStaticLocal()) {
    Rewriter.InsertTextBefore(Decl->getBeginLoc(), "static ");
  }
}
```

Figure 4.10: Code to change variable declarations matched by Figure 4.9 into static.

Based on the resolved calling tree (Subsection 4.5.1), a `talk` (or `read`) function is transformed into either a C function call or a continuation. To pass data around, the signatures of layer functions need to be transformed, which is performed by `LayerFunctionRewriter`. The AST matcher is a single

`functionDecl`. For each layer, the input and output fields are encoded with the following rules:

- Each input field in the interface becomes a function parameter (passed by value).
- Each output field becomes a function parameter that is passed by the pointer.
- All these parameters are encoded as:
  `_<from layer>To<to layer>_<field name>`.

Figure 4.11 shows an example on the right side of the transformed function signature of `CTransaction` (Controller Transaction) from the void-returning no-parameter layer function in ESM, assuming `CEepDriver` is its caller in the calling tree. For clarity, the interface between `CTransaction` and `CEepDriver` is shown on the left.

ESI

```
interface
    <CTransaction, CEepDriver>
  <= {
    i32 action;
    i32 addr;
    i32 length;
    i32 data[16];
  },
  => {
    i32 res;
    i32 length;
    i32 data[16];
  }
};
```

Promela

```
void CTransaction(
  int _CEepDriverToCTransaction_action,
  int _CEepDriverToCTransaction_addr,
  int _CEepDriverToCTransaction_length,
  intArray16
    _CEepDriverToCTransaction_data,
  int* _CTransactionToCEepDriver_res,
  int* _CTransactionToCEepDriver_length,
  intArray16*
    _CTransactionToCEepDriver_data);
```

Figure 4.11: Interface between `CTransaction` and `CEepDriver` and the corresponding transformed signature of `CTransaction`, assuming `CEepDriver` is its caller in the calling tree.

Also, the function calls of `talk` and `read` need to be transformed. To match these function calls, we use the AST matcher shown in Figure 4.12. At the outermost, it matches a function (denoted as `enclosingFunc`) that has a function body (not only a declaration). The function body should contain an assignment operation (denoted as `assignOp`) that assigns the result of a function call (denoted as `call`) to a variable (denoted as `var`).

Figure 4.13 shows an example of transforming a `talk` function into a C function call or continuation, adhering to the transformed function signatures. Similarly, Figure 4.14 shows the transformation of a `read` function.

After the number of continuations in each layer function is determined, a switch statement is inserted at the beginning of the function. This step is per-

```
functionDecl(
  hasBody(
    stmt(
      forEachDescendant(
        binaryOperator(
          hasOperatorName("="),
          hasLHS(declRefExpr().bind("var")),
          hasRHS(callExpr().bind("call"))
        ).bind("assignOp")
      )
    )
  )
).bind("enclosingFunc");
```

Figure 4.12: Clang AST matcher to match `talk` and `read` function calls.

ESM

```
v = ThisTalkOther(x, y)
```

Transformed Direct Call

```
Other(x, y, &v.a, &v.b);
```

Transformed Continuation

```
*_ThisToOther_x = x;
*_ThisToOther_y = y;
_continuation_pos = N;
return;
_continuation_N:
v.a = _OtherToThis_a;
v.b = _OtherToThis_b;
```

Figure 4.13: Example of transforming a `talk` function call into a C function call or continuation. `N` is a newly allocated continuation index.

ESM

```
v = ThisReadOther()
```

Transformed Direct Call

```
Other(0, 0, &v.a, &v.b);
```

Transformed Continuation

```
v.a = _OtherToThis_a;
v.b = _OtherToThis_b;
```

Figure 4.14: Example of transforming a `read` function call into a C function call or continuation. For a function call, the input arguments are either 0 (for built-in types) or `(T){}` for array wrapper type `T`. They should be ignored by the callee.

formed by `LayerFunctionRewriter`, after `TalkAndReadRewriter` determines the number of continuations.

Finally, since layers are making direct function calls to each other, forward declarations of the callees are inserted. This step is also performed by `LayerFunctionRewriter`, after `TalkAndReadRewriter` determines the functions being called in the layer.

4.5.3 Customized ESM Preprocessor

As mentioned in the last subsection, changes made by the rewriters are stored in a temporary buffer and flushed all at once at last. This approach works if there is only one input ESM file and it does not contain C preprocessor derivatives. However, given that ESM allows C preprocessor derivatives, a significant challenge arises.

Consider the `#include` derivatives. When an ESM file includes another file and a rewriting change occurs in that included file, the text range that Clang Rewriter receives is in another file (Clang lexer and parser preserve the include hierarchy). In that case, we cannot simply flush the change, which is not in the main input file (and therefore not reflected in the output file) but in another file.

One may consider making ESMC output multiple files and rewrite the `#include` derivatives to make them point to the corresponding output file. However, this solution cannot handle the case of *rhombus dependency*. As a concrete example, in our I^2C stack, the Controller Byte (CByte) layer and the Responder Byte (RByte) layer are essentially the same state machine. For maintainability, they include a shared function body from another file, but rename `talk` and `read` functions with macros.

The ESM files are syntactically and semantically correct. However, when performing the rewriting, the problem occurs. Two sets of changes are applied on the same file `Byte.inc.esm`.

To address this challenge, one may consider running the C preprocessor prior to ESMC to unfold all derivatives. However, this approach is not optimal. C preprocessor not only expands `#includes` of ESM files, but also those for system headers like `stdio.h`, as well as other derivatives like `#if`. Note that code generated by ESMC may target a different platform. If that is the case, users need to use the preprocessor of the cross-compiler with all compilation flags set to correctly expand the code. If the code is to be moved to the target platform and compiled natively there, this would involve a cumbersome process of moving files back and forth. Even worse, by expanding all macros, we forfeit the ability to hide code from ESMC but expose them to the final compiler (such as calls to external libraries, which ESMC does not understand)—all code in `#ifs` are already expanded.

Our approach is to implement a customized preprocessor that only expands `#includes` and lets the rewriters work on the preprocessed file. Firstly, it addresses the `#include` problem, including the rhombus dependency situation. The source files can still be split into individual files for better maintainability. After preprocessing, the output file is self-contained in a single file. At the same time, other derivatives like `#ifs` remain intact, as long as they are not in the range of parts to be replaced, such as in the middle

Controller Byte Layer

```
void CByte() {
  PREAMBLE_CByte
#define talkSymbol CByteTalkCSymbol
#define readSymbol CByteReadCSymbol
#define talkTransaction CByteTalkCTransaction
#define readTransaction CByteReadCTransaction

  CSymbolToCByte s;
  CTransactionToCByte t;
#include "Byte.inc.esm"

#undef talkSymbol
#undef readSymbol
#undef talkTransaction
#undef readTransaction
}
```

Responder Byte Layer

```
void RByte() {
  PREAMBLE_RByte
#define talkSymbol RByteTalkRSymbol
#define readSymbol RByteReadRSymbol
#define talkTransaction RByteTalkRTransaction
#define readTransaction RByteReadRTransaction

  RSymbolToRByte s;
  RTransactionToRByte t;
#include "Byte.inc.esm"

#undef talkSymbol
#undef readSymbol
#undef talkTransaction
#undef readTransaction
}
```

Figure 4.15: ESM files of Controller Byte layer (CByte) and Responder Byte layer (RByte). They are essentially the same state machine. `Byte.inc.esm` use the aliases for `talk` and `read` for polymorphism.

of a `talk` or `read` call. By default, only `#includes` with double quotes and pointing to a file that has `.esm` extension will be expanded. Other include files like system headers are not processed. However, there is also an option to expand all `#includes` to make the file more self-contained.

In addition, for completeness and usability, the customized preprocessor outputs *line derivatives*. Line directives, also known as line markers or line pragmas, are preprocessor directives that control how the source code locations are interpreted by the C/C++ compilers and debuggers. They are used to track the origin of the source code, which is essential for producing accurate diagnostics, such as error messages and warnings, as well as for

generating debugging information. There are two common types of line derivatives: one starts with `#line`, and the other starts with `#`, which is used by Clang. It has the following format.

```
# lineno "filename" flag
```

- `lineno`: A positive integer representing the line number in the source file.
- `filename`: An optional string specifying the source file's name.
- `flag`: An optional integer flag that provides additional information about the directive.

If the flag is omitted, the line derivative changes the source location directly. Otherwise, it has one of the following values:

- `1`: Indicates the start of a new file or a header file inclusion. The preprocessor pushes the current file and line number onto a stack.
- `2`: Indicates the end of a file or a header file inclusion. The preprocessor pops the previous file and line number from the stack, returning to the prior source location.
- `3`: Indicates that the following text comes from a system header file, which affects diagnostics and warnings.

C preprocessor has the ability to output line derivatives. Our customized ESM preprocessor also implements such a feature. It is useful for debugging. For example, when there is a compile-time error, the compiler can follow the line derivatives and point back to the original ESM file. The same works for debuggers, which are able to map the current instruction back to the corresponding ESM file.

While the customized preprocessor solves the `#include` problem, for files that are not inlined (such as accessory header files), there is still an include path problem. If the output file is not in the same directory as the input file, the compiler can have problems finding the file being included. According to the GCC manual[2], line derivatives do not change the include paths. The suggested solution is to add the original directory to include paths explicitly using the `-I` option. ESMC adopts this solution. When inlining files, the preprocessor collects a set of paths where the files are located and reports them to the user. The user can later use them to instruct the compiler to find included files. For the whole system compilation (discussed below), these paths are passed to the generated Makefile automatically. Alternatively, the user can instruct the preprocessor to inline all files, not only ESM files. In

[2] `https://gcc.gnu.org/onlinedocs/gcc-4.1.2/cpp/Search-Path.html`

that case, the output file is more self-contained (unless there are `#includes` that are invisible to ESMC but visible to the compiler with macros).

The customized preprocessor can also be run separately, allowing the user to inspect what code is actually being rewritten by the C backend.

4.6 HDL Backend

ESMC features generating HDL from the ESM specifications for constructing hardware drivers. As an extension to Störzbach's work, ESMC can generate not only the full stack, but also parts of it. Furthermore, the new HDL backend employs LLVM IR as the input, which allows us to integrate the arsenal of existing LLVM optimization passes into the pipeline. While these optimization passes are originally designed for optimizing software execution, our later experiments show they help reduce the resource utilization on FPGA implementation significantly.

The core idea of the HDL backend is to convert basic blocks (labels) in LLVM IR into states in the FSM described by the HDL code. Instructions in each block are translated into sequential logic, which is further implemented as combinational logic. This idea is similar to the compiler by Störzbach. However, Störzbach's compiler uses labels at the source code level, where control flow statements like `if` and `while` may still exist in the states. In contrast, at the IR level, all control flow structures have been flattened as labels and branches.

Arithmetic operations have their HDL equivalences and can be completed in the same cycle. However, `talk` and `read` require special handling. ESMC, succeeding Störzbach's compiler, also employs the handshaking protocol (Subsection 2.5.1) for inter-layer communications.

The handshaking protocol requires intermediate states. Störzbach's compiler uses common states for all `yield`s and `call`s in the same layer, as discussed in Subsection 2.3.5. ESMC takes a different approach: generate intermediate states for each `talk` or `read`. This approach may introduce more intermediate states, linear to the number of `talk`s and `read`s in the layer. However, all state transfers are deterministic at compile time, which enables optimizations like Xilinx Vivado FSM auto state encoding.

Figure 4.16 illustrates the state transfer of `talk` (①) and `read` (②) in ESMC. Three additional states are created for a `talk` and two for a `read`. To create those intermediate states, an IR manipulation is needed (Subsection 4.6.3), creating extra basic blocks that correspond to these states. Later in the process, the `talk` and `read` function calls are translated into operations in these states.

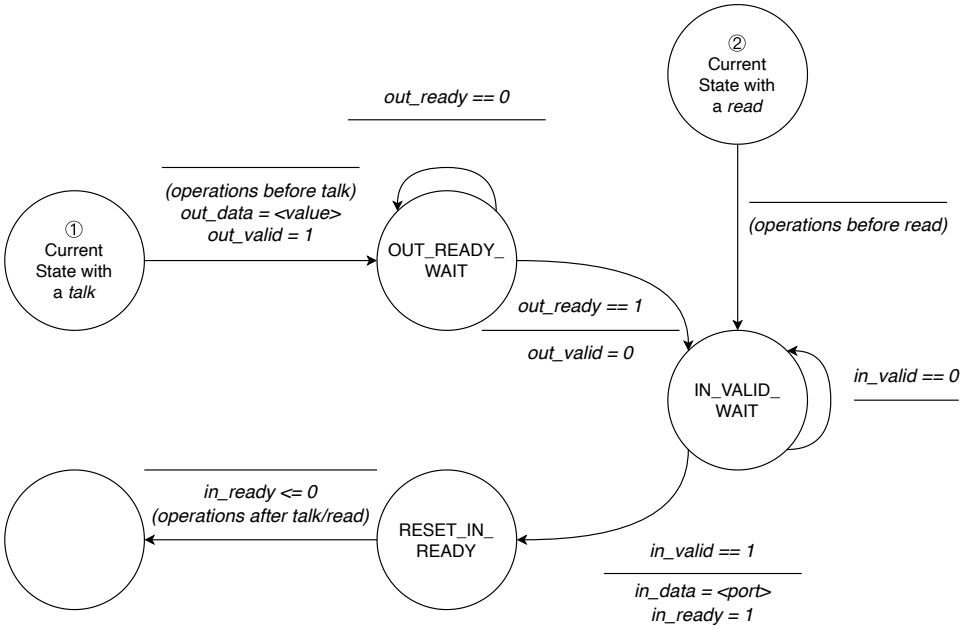


Figure 4.16: State transfer (Mealy machine) for a `talk` (①) or a `read` (②) in ESMC. State and signal names are simplified for clarity.

In this section, we present the details of the HDL backend in the order of the workflow.

4.6.1 Pre-Transformation

As shown in Figure 4.1, the input ESM file first undergoes the preprocessor, the lexer, and the parser, into Clang AST. However, unlike the other two backends that directly operate on the Clang AST, the HDL backend enters a phrase called *pre-transformation*.

In this phase, a variation of the C rewriter is used to rewrite the input ESM code. A similar rewriting as in Subsection 4.5.2 is applied to the code, except that every `talk` or `read` function is transformed to a direct C function call with no return value but input and output fields expanded as function arguments. With such a transformation, the LLVM IR generated later can be greatly simplified.

In their original form, `talk` and `read` functions return entire structs. Dealing with structs in IR can be intricate as it involves a number of pointer operations, which is not the advantage of HDL. In addition, transforming local variables to global static variables simplifies the IR as well. Otherwise, the backend would need to handle IR intrinsics like `llvm.lifetime.start`.

The pre-transformed code is fed into the preprocessor, the lexer, and the

parser again into another Clang AST.

4.6.2 LLVM IR Generation and Optimization

Once the Clang AST of the pre-transformed code is acquired, it is fed into the Clang Codegen pipeline to generate LLVM IR. Subsequently, the IR is optimized with the existing Clang/LLVM optimization passes. The optimization pipelines are encapsulated in `llvm::ModulePassManager`. Similar to Clang, ESMC allows the user to specify the optimization levels (O2, O0, etc.), but with O2 as the default configuration.

4.6.3 LLVM IR Manipulation for `talk`s and `read`s

As discussed above, the handshaking protocol requires extra cycles to operate on the ready and valid signals. Those extra cycles are reflected as extra states in the IR. Therefore, before going into the phase of generating HDL, a customized IR pass is executed to create these intermediate states. On every call instruction corresponding to a `talk` or `read`, the basic block is split three times after the call. The three new basic blocks are marked and passed to the next phase.

4.6.4 LLVM IR to Verilog

The final but most crucial step, is to transform the IR to HDL. The component for this phase is called IRTranscoder. We chose Verilog as the backend language. An out-of-the-box Verilog AST framework verilogAST-cpp[3] are used with a few modifications.

IRTranscoder operates on IR functions (which are Efeu layers). For each function, a Verilog module is created. An example is shown below. The module has ports for a clock signal and a reset signal, plus ports for interfaces. Given an interface from layer `A` to layer `B`, the Verilog module of `A` contains output ports of data fields, output valid signal, and input ready signal. Module `B` has the same ports with reversed directions. Similarly, the twin interface from `B` to `A` is encoded in the same way.

Inside the module, there is a single `always @(posedge clk)` block. The block contains the synchronous reset logic and the state machine.

[3] `https://github.com/leonardt/verilogAST-cpp`, accessed 2023-08-31.

LLVM IR

```
define void @Layer() #0 {
entry:
  ; some code
  br label %LABEL0

LABEL0:
  ; some code
  br label %entry
}
```

Verilog

```
module Layer (
    input clk,
    input rst,
    // data, valid, ready ports
);
parameter S_ENTRY = 1'd0;
parameter S_LABEL0 = 1'd1;

reg state;
// reg of variables

always @(posedge clk) begin
  if (!rst) begin
    state = S_ENTRY;
    // reset variables
  end else begin
    case (state)
    S_ENTRY : begin
      // some code
      state = S_LABEL0;
    end
    S_LABEL0 : begin
      // some code
      state = S_ENTRY;
    end
    endcase
  end
end
endmodule
```

LLVM IR uses the Static Single Assignment (SSA) representation. Each variable, either local or global, is only assigned once. Variables have *types*, such as void, integer, and pointer. When translating LLVM to Verilog, variables are translated into registers.

The void type represents nothing, thereby void-typed variables are ignored when translated to Verilog.

Non-void variables with non-pointer types are translated to registers (vectors) with the same bit sizes. The MSB is the left index and the LSB is the right index. For example, `i32` in LLVM IR corresponds to `[31:0]` in Verilog. Structs are tightly packed into vectors with no paddings between fields.

While ESM disallows pointers, the generated IR can still contain pointers. For example, the return types of the `alloca` instruction and the `getelementptr` instruction as well as global variables, are pointers. The HDL backend supports one-level pointers, that is, pointers with their dereferenced types being non-pointers. Variables with pointer types are translated to registers (vectors) with the size of the *dereferenced types*. Nested pointers are not supported.

With all SSA variables transformed into registers, the number of registers can be large. However, a large portion of them are not actually implemented as registers on FPGAs. For instance, if all reads of a register are in the same cycle as its assignment (in the same state in term of the FSM, or in the same basic block in term of the IR), EDA tools like Xilinx Vivado are capable of implementing them as wires without actually creating registers.

In the remaining part of this subsection, we present the translating rules of the supported LLVM IR instructions and intrinsics. In term of presentation, we show the LLVM IR code on the left and the corresponding Verilog on the right. Placeholders are wrapped in < and >, such as `<op>`. On the Verilog side, `|expr|` represents the processed `expr`, evaluating to the corresponding register (vector) if `expr` is a variable, the corresponding Verilog operator if `expr` is an operator, or the corresponding constant representation if `expr` is a constant, etc. `||<pointer>||` represents the special handling for pointer `<pointer>`, referring to the actual storage referred by the pointer rather the pointer itself.

`alloca` Instructions The `alloca` instruction allocates space on the stack. It returns a pointer to the stack slot. As discussed above, a register (vector) is created for the pointer with the size of the dereferenced types. No operation is generated in place for the `alloca` instruction.

Branch Instructions Branch instructions are translated into assignments to the state variable.

LLVM IR

```
br label <dest>
br i1 <cond>, label <iftrue>,
              label <iffalse>
```

Verilog

```
state = |<dest>|
state = |<cond>| ? |<iftrue>| :
                    |<iffalse>|
```

Comparison Instructions Comparison instructions are translated to the corresponding comparison operations in Verilog.

LLVM IR

```
<result> = icmp <cond> <ty>
    <op1>, <op2>
```

Verilog

```
|<result>| = |<op1>| |<cond>|
    |<op2>|
```

`|<op1>|` and `|<op2>|` is wrapped with `$signed` if `<cond>` is signed.

Load and Store Instructions Load and store instructions are translated to assignments. Note that the width of the operation does not depend on the dereferenced type of `<pointer>`, but on type `<ty>`. Therefore, `<ty>` must be taken into consideration when processing `<pointer>`.

LLVM IR

```
<result> = load <ty>, ptr
   <pointer>
store <ty> <value>, ptr
   <pointer>
```

Verilog

```
|<result>| = ||<pointer>||
||<pointer>|| = |<result>|
```

||<pointer>|| takes the whole or a part of the variable referred by the pointer based on <ty>.

Switch Instructions Switch instructions are translated to cases in Verilog.

LLVM IR

```
switch <intty> <value>, label
   <defaultdest> [ <intty>
   <val>, label <dest> ... ]
```

Verilog

```
case(|<value>|)
  default: state =
   |<defaultdest>|
  |<val>|: state = |<dest>|
  ...
endcase
```

Select Instructions Select instructions are translated into ternary operators in Verilog.

LLVM IR

```
<result> = select <selty>
   <cond>, <ty> <val1>, <ty>
   <val2>
```

Verilog

```
|<result>| = |<cond>| ?
    |<val1>| : |<val2>|
```

Only <selty> of i1 is supported.

Zero or Signed Extensions Zero or signed extensions are translated to concatenation operators in Verilog by prepending 0 or the MSB.

LLVM IR

```
<result> = zext <ty> <value>
   to <ty2>
<result> = sext <ty> <value>
   to <ty2>
```

Verilog

```
|<result>| = { |sizeof <ty2> -
    sizeof <ty>|'b0, |<value>| }
|<result>| = { |sizeof <ty2> -
    sizeof <ty>| { |<value>| [
    |sizeof <ty> - 1| ] },
    |<value>| }
```

Truncate Instructions Truncate instructions are translated to assignments by taking the LSBs.

LLVM IR

```
<result> = trunc <ty> <value>
   to <ty2>
```

Verilog

```
|<result>| = |<value>|[|sizeof
    <ty2> - 1|:0]
```

Binary Operations Binary operators in LLVM IR are translated to the corresponding operators in Verilog. Since ESM does not support floating-point operators, they do not exist in the resulting IR.

LLVM IR

```
<result> = <binop> <ty> <op1>,
    <op2>
```

<binop> can be add, sub, mul, udiv, sdiv, shl, lshr, ashr, and, or, xor, but no floating-point ops

Verilog

```
|<result>| = |<op1>| |<binop>|
    |<op2>|
```

|<op1>| and |<op2>| may be warpped with `$signed()`

PHI Instructions The PHI instruction is used to implement the φ node in the LLVM SSA graph. They are placed at the beginning of the basic block. It returns the value associated with the predecessor basic block that is executed just prior to the current block. When translated to Verilog, no operation is generated in place. Instead, at the end of each predecessor block, an assignment is placed.

LLVM IR

```
<result> = phi <ty> [ <val0>,
   <label0> ], ...
```

Verilog

```
At the end of each <label>:
|<result>| = |<value>|
```

LLVM `memcpy` Intrinsics `llvm.memcpy` intrinsics are translated to assignments with the copy size statically calculated. The size must be a constant at compile time.

LLVM IR

```
call @llvm.memcpy.p0.p0.i32/i64
   (ptr <dest>, ptr <src>,
    i32 <len>, i1 <isvolatile>)
```

Verilog

```
||<dest>|| = ||<src>||
```

LLVM `memset` Intrinsics `llvm.memcpy` intrinsics are translated to assignments by replicating the 8-bit value. The length to assign must be a constant at compile time.

LLVM IR

```
call @llvm.memset.p0.i32/i64
   (ptr <dest>, i8 <val>,
    i32 <len>, i1 <isvolatile>)
```

Verilog

```
||<dest>|| =
   {|<len>|{|<val>|}};
```

`getelementptr` Operators/Instructions `getelementptr` can be used as a standalone instruction or an operand of another instruction. Its only use case in the IR generated from ESM code is for accesses into structs or arrays. IRTranscoder takes the register (vector) corresponding to the input pointer and calculates the offset by following the structs and/or array.

`talk` and `read` Function Calls As discussed above, `talk` and `read` function calls are implemented with the handshaking protocol. The intermediate states have been created by the IR manipulation passed discussed in Subsection 4.6.3.

In IRTranscoder, the corresponding function call is translated into operations into states.

LLVM IR

```
call void <fn> (<out-ty> <out-val>, ..., ptr <in-ty>, <in-ptr>,
    ...)
```

The call must be a valid `talk/read` function with correct params.

Verilog

```
|CURRENT_BASIC_BLOCK|: begin
  (previous code)
  |out-data-port| = |<out-val>|; (if talk)
  |out-valid-port| = 1'b1;       (if talk)
  state = |OUT_READY_WAITING|;   (if talk, or |IN_VALID_WAITING|
    if read)
end
|OUT_READY_WAITING|: begin       (if talk, or no this state if
    read)
  if (|out-ready-port| == 1'b1) begin
    |out-valid-port| = 1'b0;
    state = |IN_VALID_WAITING|;
  end
end
|IN_VALID_WAITING|: begin
  if (|in-valid-port| == 1'b1) begin
    ||<in-ptr>|| = |in-data-port|;
    |in-ready-port| = 1'b1;
    state = |RESET_IN_READY|;
  end
end
|RESET_IN_READY|: begin
  |in-ready-port| = 1'b0;
  (remaining code)
end
```

Intermediate states are created ahead of time.

4.7 Integration with Build Systems

ESMC acts as a compiler. For complicated systems, it is a common practice to organize source files into a hierarchy and employ build systems like Make or CMake. In the context of the I^2C stack, layer specification files are shared by multiple complication targets to create different configurations. With the end-user usability in mind, Efeu is designed to be easily integrated with build systems. In this section, we discuss those features.

4.7.1 Use of Preprocessor Derivatives

The ESM language allows C preprocessor macros, which enables users to create systems with flexibility.

Modularity The incorporation of `#include` and `#define` facilitates modular designs and helps reduce duplication. For example, as already shown in Figure 4.15 above, the same state machine can be reused in multiple layers by using these two preprocessor derivatives. Layers can also be assembled using `#include`. For example, Figure 4.17 shows the assemble of a top-down controller stack, with the World layer as the calling point and the Electrical layer bridging to the GPIO driver to perform bit-banging on the I^2C bus.

```
1  #include "boilerplate/CWorld-entry.esm"
2  #include "layers/CEepDriver.esm"
3  #include "layers/CTransaction.esm"
4  #include "layers/CByte.esm"
5  #include "layers/CSymbol.esm"
6  #include "boilerplate/CElectrical-gpio.esm"
7
8  #if defined(__ESMC_C__) && defined(__GNUC__)
9  #include <stdio.h>
10
11 int main() {
12     /* some code */
13 }
14 #endif
```

Figure 4.17: ESM code of a top-down controller stack, with CWorld (Controller World) as the calling point and CElectrical of a bridge to GPIO.

Polymorphism Polymorphism can be achieved by using `#define`. For example, the RTransaction (Responder Transaction) layer uses macro `RESP_ADDR` as the responder I^2C address. By creating multiple instances of the responder and specifying different values for `RESP_ADDR`, a multi-responder system can be assembled.

Conditional Compiling Conditional derivatives like `#if` allow conditional compiling. Similar to Clang and GCC, ESMC defines macros to differentiate different backends.

- `__ESMC_PROMELA__` is defined when running the Promela backend.
- `__ESMC_C__` is defined when running the C backend and *should* be defined when running the C compiler on the generated C code (which is controlled by the end-user when invoking the C compiler). The main reason is to allow native-C boilerplate code to be wrapped in `#if defined(__ESMC_C__)&& defined(__GNUC__)`, where the latter is defined by Clang and GCC. Asserting on only `__GNUC__` is not sufficient, as the Promela pan compiler also invokes Clang or GCC.
- `__ESMC_HDL__` is defined when running the HDL backend.

With conditional compiling, boilerplate code can be written directly in ESM files, like Lines 8–14 in Figure 4.17.

4.7.2 Auto Dependency Generation

An important aspect of the compilation process, especially in large projects, is managing dependencies between source files. Dependencies ensure that when a file is changed, all files that rely on it are recompiled to reflect those changes. This is particularly crucial with C family languages where header files are often used across multiple source files. To help manage these dependencies, GCC and Clang both provide the `-MD` command line option. The option instructs the compiler to generate a list of dependencies into a file, the depfile, by scanning through the source code. Such functionality is built into the preprocessor. Build systems like Make and Ninja make use of the information to ensure correct and efficient incremental builds.

Built upon Clang/LLVM, ESMC inherits such a feature. The same `-MD` option instructs ESMC to generate a depfile for the input ESM source file. The user can also change the filename of the depfile using `-MF` option, in the same way as GCC and Clang.

4.7.3 CMake Build System Integration

To further ease the work of using ESMC in the CMake build system, several CMake functions are created for the end-users to define compile targets with auto dependency resolution. For example, the CMakeLists code piece in Figure 4.18 defines a target called `top-down-c1-gpio` using the ESMC C backend that compiles the specified ESI and ESM files, and generates executable using the default C compiler after combining with the boilerplate code. The calling point is `CWorld`. For the complete usage of those CMake functions, please refer to the ESMC user manual.

```
1 esm_add_c_executable(top-down-c1-gpio
2     ESI i2c.esi
3     ESM assemble/top-down-c1-gpio.esm
4     ADDITIONAL_SRC boilerplate/console.c boilerplate/i2c_gpio.c
5     INCLUDE_DIRS boilerplate
6     CALLING_POINT CWorld)
```

Figure 4.18: A CMake code piece to add an executable target using ESMC.

4.8 Whole System Compilation

Advancing beyond the previous works, ESMC supports the whole system compilation, allowing the composition of software and hardware components

into a single stack. The system is specified by a YAML file. Figure 4.19 shows an example of such YAML files. The file specifies the following configurations.

- The configuration description.
- Working directory for all relative paths in the configuration.
- ESI filename and the corresponding generated header.
- Layers and their corresponding ESM files.
- Components. Each component is set to run against a backend. Its configuration includes the set of layers and the backend-specific options.
- Interfaces between components. Each interface has a type, an origin layer in one component, and a destination layer in another component, as well as interface-specific options.

Whole system compilation not only allows mixing heterogeneous components into a driver, but also takes care of handling interface generation between components. Conceptually, generating interfaces between heterogenous components is just an another way of implementing the `talk` function, like call and continuation between C layers, and the handshaking protocol between hardware layers.

Currently, the only interface between heterogenous components supported by ESMC is the memory-mapped IO (MMIO) AXI Lite interface, between a C component and a HDL component. However, one can imagine more interfaces to be supported by ESMC. For instance, there can be a way to `talk` via inter-process communication mechanism, or across the VM-host boundary through virtual devices to implement software drivers with strong isolation.

4.8.1 MMIO-AXI Lite Interface

ESMC is capable of generating the MMIO-AXI Lite interface between a C component and an HDL component. Figure 4.20 illustrates the architecture of a system with one C component and a HDL component, connected with the MMIO-AXI Lite interface. Between the two layers that cross the software-hardware boundary, their interfaces, plus the ready and valid signal for the handshaking protocol, are mapped to AXI Lite registers with different offsets. For clarity, in the following paragraphs, we call the software part as MMIO driver, and the hardware part the AXI Lite driver.

Mapping Interfaces to Registers With the current implementation, AXI Lite with 32-bit bus width is supported. Each register is 32 bits wide, or 4 bytes in the memory-mapped address space. The first register with offset 0 is

```
description: "Top-down EEPROM controller driver with CWorld to
    CEepDriver in software and CTransaction to CSymbol in
    hardware, connected with AXI Lite"
rootDirectory: ".."
esi: i2c.esi
esi-header: i2c.esi.h
layerFiles:
  CWorld: boilerplate/CWorld-entry.esm
  CEepDriver: layers/CEepDriver.esm
  CTransaction: layers/CTransaction.esm
  CSymbol: layers/CSymbol.esm
  CByte: layers/CByte.esm
components:
  SW:
    backend: c
    layers:
      - CWorld
      - CEepDriver
    callingPoint: CWorld
    outputType: executable
    outputFilename: top-down-CTransaction-IP-SW
    extraSourceFiles:
      - boilerplate/console.c
      - boilerplate/main-console-CWorld.c
  HW:
    backend: hdl
    layers:
      - CTransaction
      - CByte
      - CSymbol
interfaces:
  SW-HW:
    from: CEepDriver
    to: CTransaction
    type: axilite
    mmioHeader: boilerplate/mmap-io.h
    mmioSource: boilerplate/mmap-io.c
```

Figure 4.19: Example of the YAML file that describes the system for ESMC whole system compilation.

reserved for status checking and resetting. If a data field in the layer interface is larger than 32 bits (for example, an array), it spans across multiple registers. If a data field is less than 32 bits, it takes the low bits of the register. The valid and ready signal follows the data fields. While they both require only one bit each, they each take up one register, mainly for ease of implementation. A more compact encoding of the interface remains a future work.

Status and Reset Register When generating the connection, ESMC computes a checksum with the names of the two connected layers and data fields

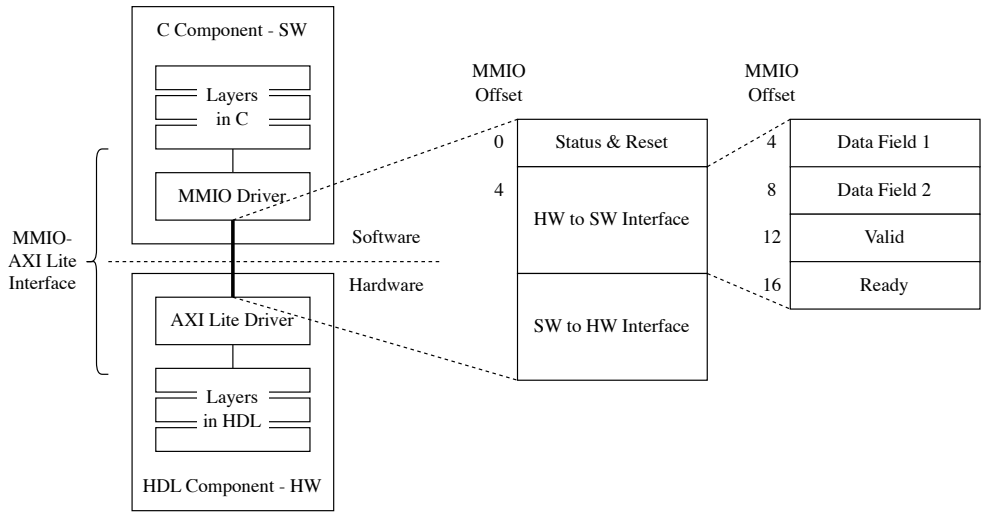


Figure 4.20: Architecture of a system with one C component and an HDL component, connected with the MMIO-AXI Lite interface.

in their interfaces. On the software side, the MMIO driver reads the status register once at initialization. If the result matches the magic number, it indicates the two sides match. A connection is then established. On the hardware side, the AXI Lite driver is able to detect the read. Subsequent reads on the status register return 0, indicating the device is already used by a program.

If the MMIO driver reads back 0 from the status register, it means there is another program that already established a connection with the AXI Lite driver. In this case, the MMIO driver has the option to reset the hardware part, by writing the magic number back to the status register. This action resets the AXI Lite driver as well as layers in hardware (assuming they are connected to the reset signal correctly in the FPGA design). Subsequently, the hardware part becomes connectable again. If the program previously connected to the hardware is still running, a conflict may arise. It is the software's responsibility to decide whether to reset and connect or to about. The AXI Lite Driver only resets itself if the magic number is written. Without knowing the magic number, other programs cannot accidentally or maliciously reset the AXI Lite driver.

AXI Lite Driver AXI Lite Driver bridges the AXI Lite interface with the layer that it connects to, providing the same ports as the layer interface (data + valid + ready ports). Connecting the data field to the memory-mapped registers is straightforward. However, the valid and ready signals require extra attention.

The handshaking protocol assumes the sender and the receiver operate in the

same clock domain. After a cycle where both valid and ready are raised, the sender needs to lower the valid signal immediately on the next clock cycle if there is no more data to send. Otherwise, the receiver treats data on the bus as the next valid packet. Similarly, the receiver needs to lower the ready signal unless it can immediately accept more data on the next cycle.

However, when crossing the software and hardware boundary, the two sides are not in the same clock domain. If the valid port is implemented as a dumb register, it is possible that the software side cannot reset it in time, resulting in the same data being transmitted multiple times. Similarly, if the software side does not reset its ready port in time, the hardware side considers it to be ready all the time and may send multiple packets that overwrite one another, resulting in data loss.

Our solution to this problem is to perform automatic resets on the hardware side in the AXI Lite Driver, which is in the same clock domain as the HDL layers. Software writing a non-zero value to its output valid port means the data in the data registers are valid *once*. If the data is consumed, the valid signal is lowered in the hardware on the next cycle. Similarly, writing a non-zero value to its output ready port means the MMIO driver is ready to accept *one* packet. Once a packet is in place, the ready signal is lowered by the hardware on the next cycle.

MMIO Driver MMIO driver consists of the implementation of the corresponding `talk` and `read` functions, adhering to the handshaking protocol. Targeting the Linux platform, the MMIO driver memory maps the registers through `/dev/mem` at initialization. With auto-resetting ready and valid registers in the AXI Lite Driver, MMIO driver only needs to assert them but does not need to deassert them.

Chapter 5

Verification

With the Promela backend in ESMC, the I^2C stack can be translated to Promela code for model checking. The controller I^2C stack written with Efeu is verified against the verifiers described in Subsection 2.3.3 using the SPIN model checker to ensure it conforms to the I^2C specification. Specifically, the composition of one controller and one responder up to a target layer (Symbol, Byte, Transaction, and EepDriver) is equivalent to the corresponding abstraction viewing from the immediately higher level. Verifiers also ensure the absence of deadlocks and livelocks in the compositions.

The experiment is performed on an Intel Xeon W-2255 CPU @ 3.70GHz machine running Ubuntu 20.04.4 LTS with 126 GB RAM. The SPIN version is 6.5.2, the latest version available at the time of writing. The verifier setup is identical to the previous works [14, 15]. Every verifier is compiled twice, one with the default SPIN pan compilation options, and the other with the Non-progress Cycle (NPC) check (discussed in Subsection 2.2.1) enabled. Output of the pan verifiers is checked for any detected issues. In addition, every verifier is executed 3 times and the run time is measured. For completeness, like previous works, we also test how layers of abstraction help speed up the verification process.

All verifiers pass. The average run times are reported in Table 5.1 and Table 5.2. Verifying the full implementation of layers up to the Transaction layer takes the most time, 39.23 seconds. Most verifiers take less than 10 seconds to run.

Table 5.1: Average Verification Run Time (s) Using the Default SPIN Options

	Full Implementation	SymbolSpec	ByteSpec	TransactionSpec
Symbol	0.08			
Byte	1.33	0.43		
Transaction	14.82	5.12	1.15	
EepDriver	9.22	3.02	0.54	0.19

Table 5.2: Average Verification Run Time (s) with the Non-progress Cycle Checks Enabled

	Full Implementation	SymbolSpec	ByteSpec	TransactionSpec
Symbol	0.10			
Byte	3.37	1.10		
Transaction	39.23	14.01	3.25	
EepDriver	24.97	8.27	1.29	0.41

Chapter 6

Evaluation on Real Devices

In this section, we present the evaluation results of the I^{2}C stack. All experiments discussed in this chapter are conducted with the following setup.

I^{2}C Controller Experiments run on the ZynqMP SoC [32] on an Enclustra XU5 module [8]. The module is placed on the PE1 Based Board [7]. Two GPIO pins connected to the SoC PL are used as the SCL and the SDA, which are routed to the PE1 baseboard and then to an IO connector.

I^{2}C Responder (EEPROM) A Microchip 24AA512 EEPROM [20] is connected to the I^{2}C bus, with its address configured to 0x50. Compared to volatile memory like RAM, EEPROMs have slower write speed. 24AA512 page writes can take up to 5 ms. Internally, the chip has a self-timed erase/write cycle to handle burst writes from the I^{2}C controller. However, during the periods when the device is busy, it may stop responding to the subsequent I^{2}C requests. In this evaluation, our main focus is the performance of the generated drivers on I^{2}C transactions, rather than the performance of the EEPROM itself. Therefore, we only use write operations in manual validations, in which the EEPROM has sufficient time to respond. In contrast, in experiments driven by programs, especially in those time-measuring experiments (Subsection 6.2.2 and Subsection 6.3.1), we only issue read requests to the EEPROM, which the device has no problem handling at high speeds.

Oscilloscope A Keysight InfiniiVision MSO-X 3024T oscilloscope [18] is connected to the SCL and the SDA lines. The oscilloscope is capable of decoding I^{2}C packets.

Software APU of the SoC runs a modified OpenBMC [5] distribution, which is based on Linux kernel 5.4.0. Software parts of the generated I^{2}C drivers

are cross-compiled to AArch64 with GCC 11.2.0 with O3 optimization.

FPGA Design Customized bitstreams are flashed into the FPGA in the SoC PL. The base FPGA design consists of components for the Enzian BMC, such as the GPIO drivers. The relevant part with the evaluation is an IP block connected to the SCL and SDA pins, either a Xilinx GPIO driver (for bit-banging drivers) or a generated I^2C driver (for combined software-hardware drivers), as shown in the bottom-right of Figure 6.1. All components are driven by the 100 MHz clock from the MPSoC. The design is synthesized and implemented using Xilinx Vivado 2020.1 with default settings.

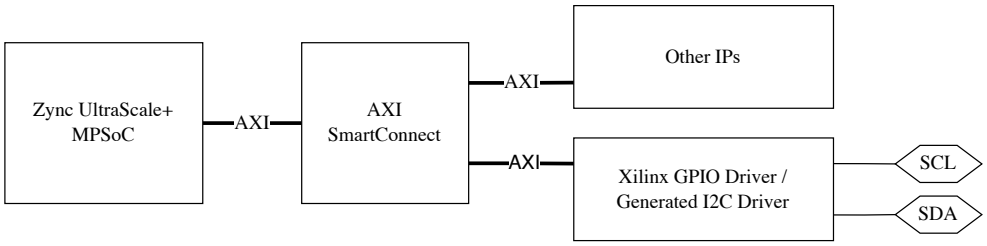


Figure 6.1: Simplified FPGA block diagram.

6.1 Baseline

To establish the baseline, we first reproduce the results in Humbel's work [14, 15], using their generated C code combined with the boilerplate code shipped with the codebase. The purpose of this experiment is to validate the functionality of the I^2C stack specification from the original framework for our test platform. The original bit-banging code, as seen in Appendix B, uses very conservative timing. There are long manually inserted intervals between SCL and SDA operations. Since the main focus of this experiment is functional correctness, while timing is less of a focus, we keep the bit-banging as the original codebase.

The I^2C stack is compiled into C code using the compiler from Humbel *et al.*. The resulting code is combined with the bit-banging boilerplate code and compiled into an executable. The Xilinx GPIO driver IP is used in the FPGA design. The executable controls SCL/SDA pins through Linux sysfs. The evaluation is conducted on the experiment platform described above.

Figure 6.2 shows the decoded I^2C transactions by the oscilloscope. They correspond to the hardcoded operations in the original top layer as shown in Figure 3.2. The first transaction is a write operation for 2 bytes starting from address 0x06, with values 32 (0x20) and 33 (0x21). The second transaction is a EEPROM read starting from address 0x06. The same values are read back.

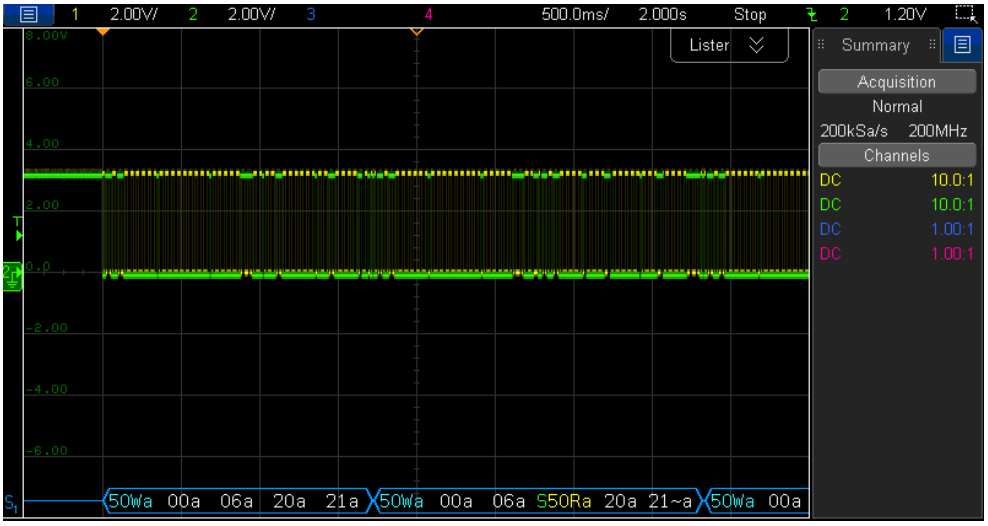


Figure 6.2: Decoded I^{2}C transactions using the generated software drivers in the previous work [14, 15]. The yellow signal is SCL. The green signal is SDA. Decoded values are shown in hexadecimal. `W`: write. `R`: read. `a`: ACKnowledged. `S`: restart.

Decoded I^{2}C transactions show that the hardcoded high-level operations undergo the stack and correctly operate the electrical bus. The transaction timing is not ideal (Figure 6.2 is displayed with the time scale of 500ms). The read operations takes 1.7 seconds and the write operation takes 2.0 seconds.

6.2 Bit-Banging Software Driver

In this section, we showcase the evaluation result on the bit-banging software-based driver written with our new framework Efeu, generated with the ESMC C backend. We first validate its functionalities by gluing it with the *unmodified* bit-banging driver code.

After that, we optimize the bit-banging driver code originating from the previous codebase by removing all manual delays. An experiment shows that even without these manually inserted delays, Linux sysfs introduces a significant overhead that prevents the driver from running at the target speed (400 kbit/s). We report the timing in the second subsection.

6.2.1 Functional Correctness

In this experiment, we validate the functional correctness of the generated software driver. Specifically, we are interested in whether the major advancement of Efeu to its predecessor works correctly: top-down architecture with an open interface at the top level. We use the *unmodified* bit-banging driver code at the Electrical layer.

The I^2C stack is compiled into C code using ESMC. The resulting code is combined with the bit-banging boilerplate code. As the stack written in Efeu now has an open interface at the top level, we write a simple shell to drive the stack with users' inputs. The combined code is compiled into an executable, which runs on the experiment platform described above. Xilinx GPIO driver IP is used in the FPGA design.

Figure 6.3 shows the log of a manual validation of the driver. Lines starting with > are input commands. Firstly, the driver is invoked to read one byte from EEPROM address 0x06 (Line 4). The operation is done successfully, indicated by the status code `ME_RES_OK`. Returned data follows. Next, the driver is invoked to read 8 bytes starting from address 0x06. Subsequently, the driver is used to read and write EEPROM with different data, length, and/or starting addresses. The results are all expected. On the last write (Line 13), the transaction is monitored by the oscilloscope. The waveform is shown in Figure 6.4. The decoded transaction matches the expected transaction—reading 8 bytes starting from the address 0x08 that matches the previously written values, showing the operations are actually performed on the electrical level.

```
 1 root@mercury-xu5-1:~# ./top-down-c1-gpio
 2 Read  command: r <addr_in_eeprom> <length>
 3 Write command: w <addr_in_eeprom> <data0> <data1> ...
 4 > r 0x06 1
 5 [CWorld Init] res: ME_RES_IDLE
 6 [CWorld] res: ME_RES_OK [0]1
 7 > r 0x06 8
 8 [CWorld] res: ME_RES_OK [0]1 [1]2 [2]3 [3]4 [4]5 [5]6 [6]7 [7]8
 9 > w 0x06 11 22 33 44 55 66 77 88
10 [CWorld] res: ME_RES_OK
11 > r 0x06 8
12 [CWorld] res: ME_RES_OK [0]11 [1]22 [2]33 [3]44 [4]55 [5]66
       [6]77 [7]88
13 > w 0x07 255 254 253 252 251 250
14 [CWorld] res: ME_RES_OK
15 > r 0x08 8
16 [CWorld] res: ME_RES_OK [0]254 [1]253 [2]252 [3]251 [4]250 [5]88
       [6]9 [7]10
```

Figure 6.3: Manual validation of the top-down controller stack. Lines starting with > are input commands. `ME_RES_OK` is the return value from the top layer, indicating the operation is done successfully. For read operations, the returned data array follows, where the numbers in `[]` are indices into the data buffer.

From the results, we can conclude that the top-down bit-banging driver functions correctly in accordance with the input instructions (a more comprehensive test requires more test cases).

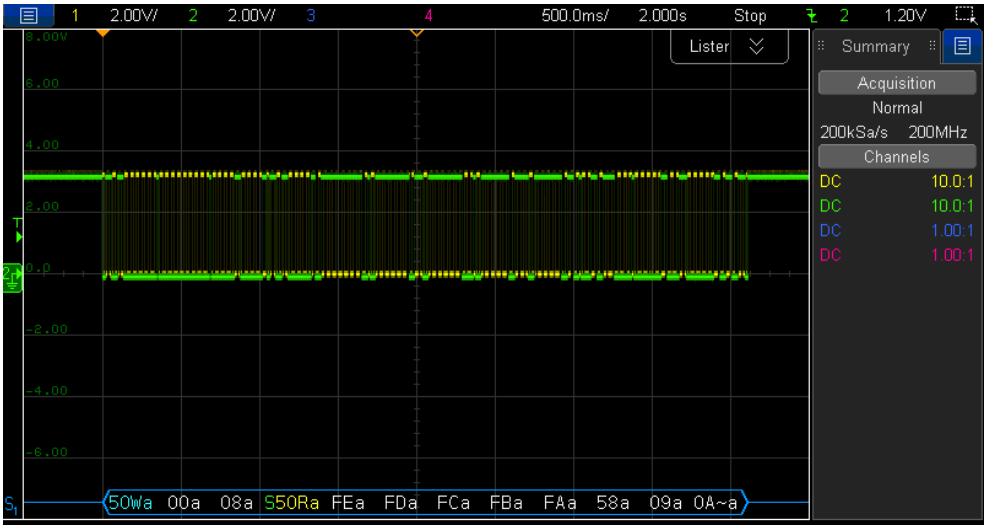


Figure 6.4: Captured waveform of the last write operation (Line 13) of Figure 6.3. The yellow signal is SCL. The green signal is SDA. Decoded values are shown in hexadecimal. `W`: write. `R`: read. `a`: ACKnowledged. `S`: restart.

6.2.2 Timing

To get a more comprehensive understanding of the overhead imposed by the bit-banging GPIO driver through Linux sysfs, we modify the code by removing all manually inserted delays and measured the I^2C transaction time. Furthermore, the test program issues commands in sequence one after another, eliminating the time spent on user interactions.

The test program issues 200 identical read commands. Each command reads 8 bytes starting from address 0x06. The Linux built-in command `time` is used to measure the execution time.

Issuing the 200 read commands takes 11.3 seconds on average across 3 runs without significant differences between runs. Time spent in the user space accounts for 2.2s, 19.6%, while time spent in the kernel is 9.1 s, 80.4%. Figure 6.5 illustrates the time breakdown. Figure 6.6 shows the captured waveform of the first few symbols of a transaction. The delay of a bit-banging operation is roughly 125 us by observing the first few cycles. The corresponding maximum clock cycle is 250 us (a cycle consists of a rising edge and a falling edge), or 4 kHz in terms of frequency, far lower than the target frequency of 400 kHz. The actual transmission frequency is even lower since SCL is not flipped on every operation in the Electrical layer. There can be consecutive high or low signals issued from the Symbol layer to the Electrical layer.

Such a result indicates that bit-banging through the Linux sysfs indeed imposes a significant overhead, preventing the generated driver from reaching

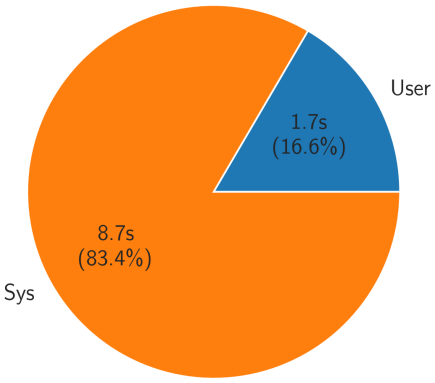


Figure 6.5: Time breakdown of issuing the 200 read commands using the bit-banging driver.

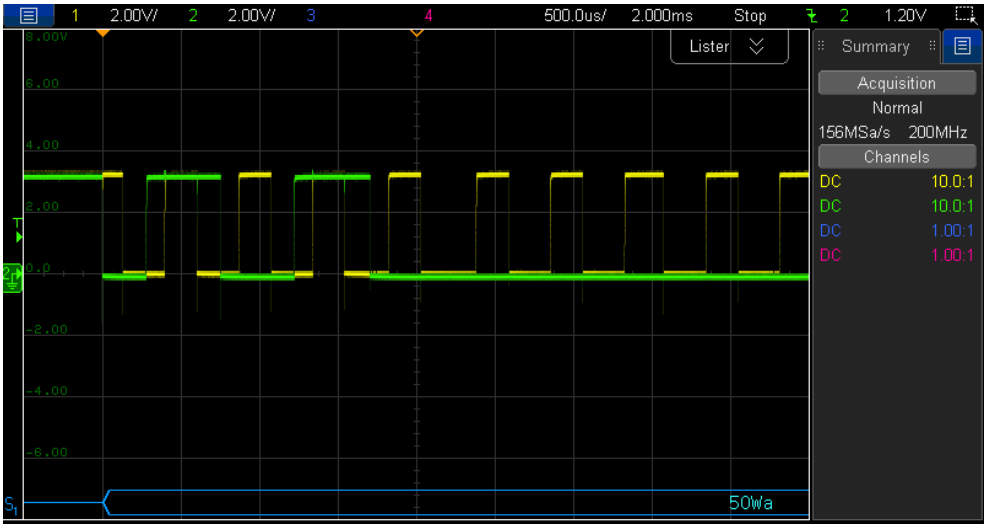


Figure 6.6: Captured waveform of the first few symbol of a transaction with the optimized bit-banging driver. The yellow signal is SCL. The green signal is SDA.

the target speed. We do not further invest the time breakdown, due to limited tools on the OpenBMC distribution running on the BMC module, which has neither profiling tools like perf or compilers to compile them. More importantly, accessing GPIO through Linux sysfs has been deprecated. The current recommended approach is using the new user-space GPIO APIs. Alternative driver design, the combined software-hardware drivers we discuss next, is also an approach to avoid the overhead of the bit-banging driver.

6.3 Combined Software-Hardware Drivers

With the whole system compilation, Efeu is able to generate combined software-hardware drivers. Specifically for the I^2C stack, we evaluate multiple drivers by choosing a split point between layers, putting the bottom part into hardware and the top part into software. We present the measurements on the timing of I^2C transactions in Subsection 6.3.1 and FPGA utilization in Subsection 6.3.2.

FPGA utilization is affected by optimizations involved in the pipeline. In Subsection 6.3.3, we present evaluations on the effect of two optimizations: the LLVM optimization passes and the Xilinx Vivado FSM optimization. We also compare the result with the driver generated in the previous work [27] as the baseline.

For completeness, we also perform the same evaluations on the Xilinx I^2C IP in the FPGA design plus the Xilinx bare metal driver in software. The Xilinx I^2C IP functions at an abstraction level similar to the Transaction layer in our stack, supporting sending variable-length transactions to the specified address. It also features additional functions like FIFO and interrupts, which are not supported by our I^2C stack. All drivers operate the I^2C bus at the High Speed Mode (400 kHz).

While splitting drivers into a high-level part and a low-level part adheres to common practice, Efeu provides flexibility far beyond these configurations. As a demonstration, we show a configuration that crosses the software-hardware boundary multiple times in Subsection 6.3.4.

6.3.1 Timing

Choosing the split point between software and hardware has an impact on the timing of I^2C transactions. In this experiment, we measure the I^2C transaction timing at the electrical level using the oscilloscope.

With each combined software-hardware driver, EEPROM operations are issued from the software side. Read operations with two different lengths are used for testing. The experiment is performed on the platform discussed previously. The two read lengths are 16 bytes and 1 byte. Each read command is repeated 3 times. We use the oscilloscope to plot waveforms of SCL and SDA (e.g. Figure 6.7). The oscilloscope is triggered at the start of the first operation and records long enough to the end of the last operation. Gaps between the three operations are identified by observing the bus being idle for a certain amount of time, and therefore the start and end times of each operation are recognized. The elapsed time is calculated as the average of the three operations.

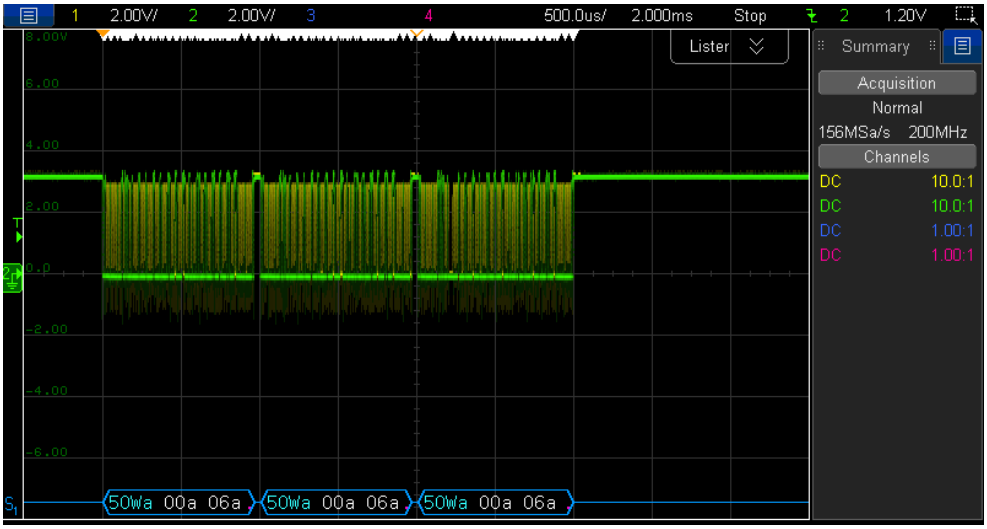


Figure 6.7: Plot waveforms of three consecutive operations of reading 16 bytes, using combined software-hardware driver split between Transaction and Byte. Gaps between operations are observable and identifiable by setting a threshold. The yellow signal is SCL. The green signal is SDA. Decoded values are shown in hexadecimal. `W`: write. `R`: read. `a`: ACKnowledged. `S`: restart.

Figure 6.8 shows the average elapsed time of reading 16 bytes. The Xilinx IP takes the least amount of time. For our drivers, the elapsed time decreases as we lower the stack into hardware. With the whole stack in FPGA (split point EepDriver), the generated driver yet takes 1.58× time for the operation compared to Xilinx IP. Reading 1 byte shows a similar trend, as shown in Figure 6.9.

The difference between Xilinx IP and EepDriver can be explained by the different encodings of BIT 0 and BIT 1. By observing the first few symbols sent by the Xilinx IP (Figure 6.10) and EepDriver (Figure 6.11), we can see that the Symbol layer is very conservative in the encoding. There is a stretching cycle after the start symbol, and each BIT 0 and BIT 1 takes two more cycle for encoding. Such behavior matches the ESM source code of the Symbol layer (we make minimal changes when porting the specification to Efeu). The specification uses such conservative encodings for reliability and possibly to accommodate non-compliant devices. Due to the time constraint, we do not investigate the effect of removing these redundant cycles. However, there is certainly a space for timing optimization.

Regarding the timing difference with different split points, one can compare the waveform of Electrical and EepDriver in Figure 6.12. Both plots are zoomed at the same scale and aligned in time. We can observe that the waveform of Electrical is more sparse, which can be explained by the overhead of handshaking crossing the software-hardware boundary. In hardware,

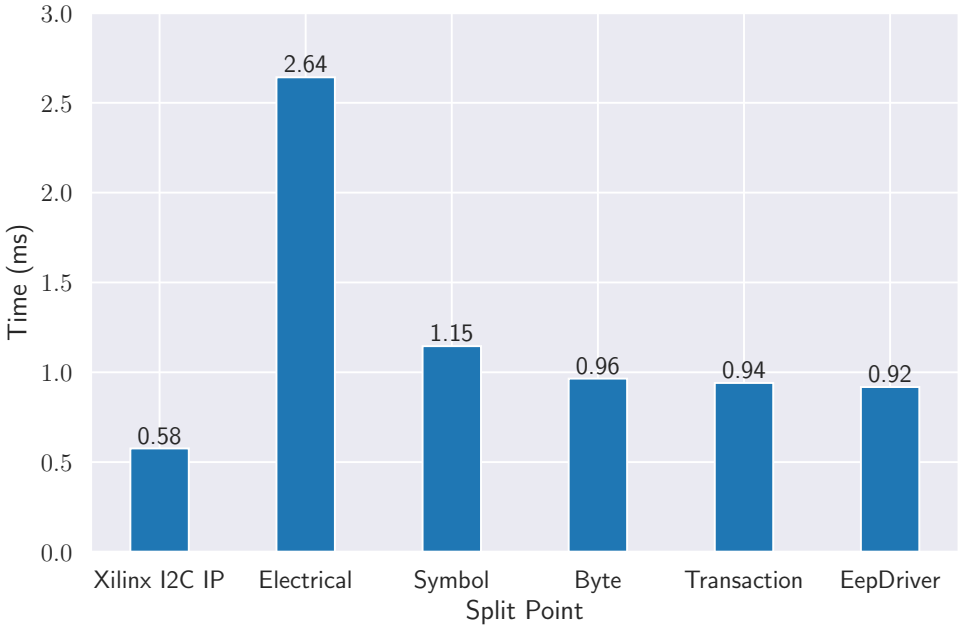


Figure 6.8: Average elapsed time of reading 16 bytes. The split point is the topmost layer in hardware, whose upper layer is the bottommost layer in software.

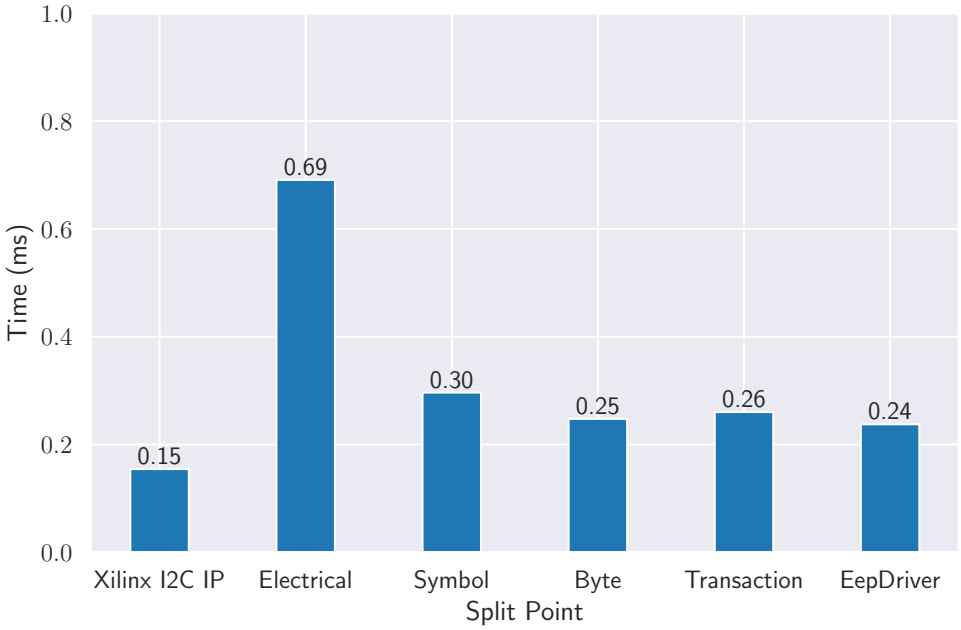


Figure 6.9: Average elapsed time of reading 1 byte. The split point is the topmost layer in hardware, whose upper layer is the bottommost layer in software.

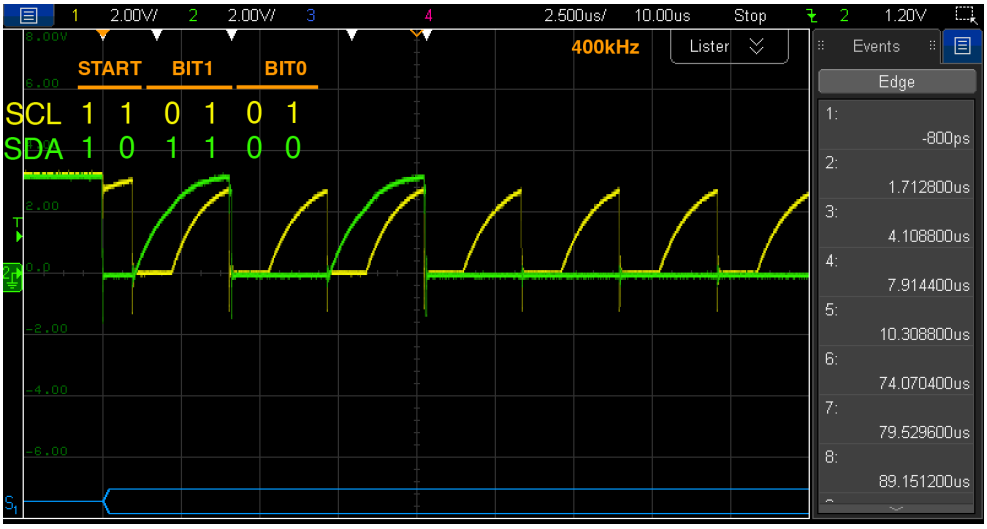


Figure 6.10: First few symbols of Xilinx IP when performing EEPROM read operations.

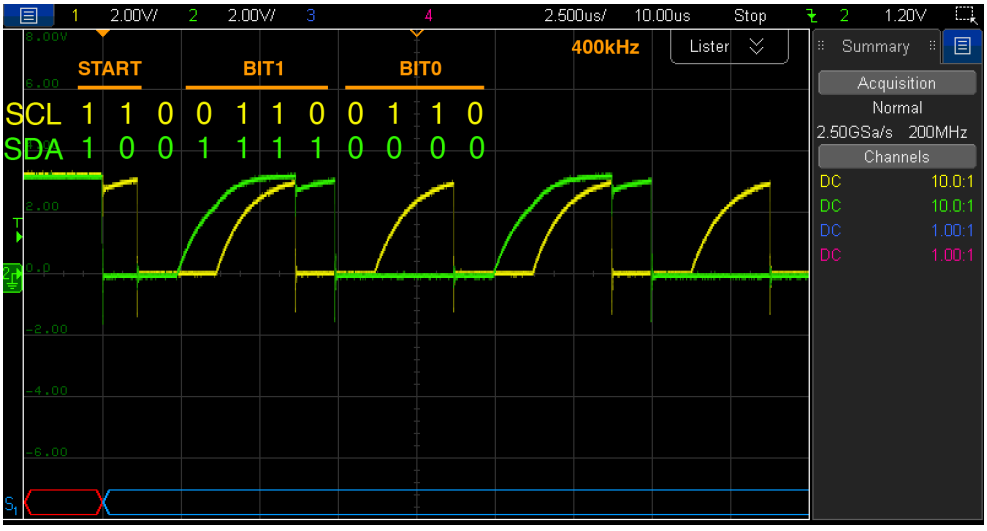


Figure 6.11: First few symbols of EepDriver when performing EEPROM read operations.

data and valid signals are supplied in the same clock cycle. In contrast, when emulating the handshaking with AXI Lite-based registers, each write to a data register or the valid register takes one cycle. If the time when the software issues the write request does not match the FPGA clock boundary, the process takes even more cycles. As a consequence, more frequent communication across the software-hardware boundary results in more time spent at handshaking.

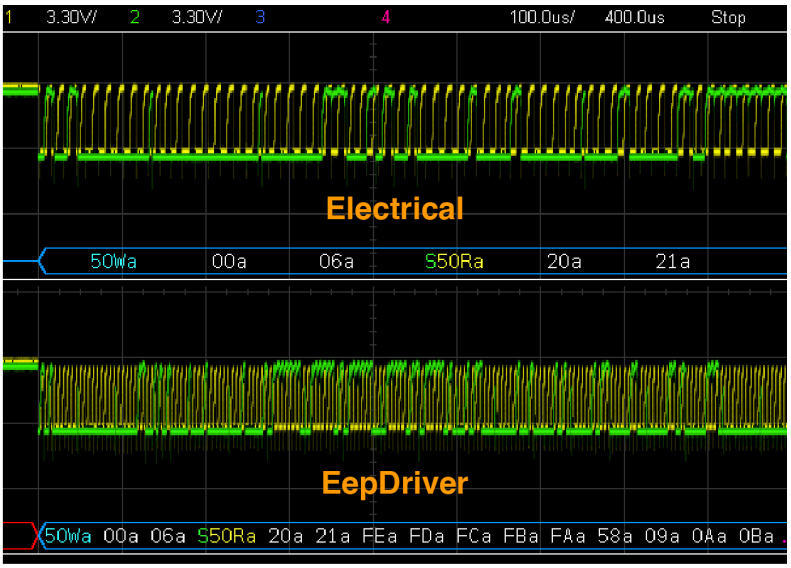


Figure 6.12: Waveforms of Electrical vs EepDriver. Both plots are scaled at 100us and aligned in time. Yellow signal is SCL. Green signal is SDA. Decoded values are shown in hexadecimal. `W`: write. `R`: read. `a`: ACKnowledged. `S`: restart.

Between layers where a small amount of data is transmitted, like between Symbol and Electrical layer, using compact data types (for example, use a single bit instead of 32-bit integer for SCL and SDA) and packing data and/or valid/ready signal into AXI Lite registers may help mitigate such overhead.

6.3.2 FPGA Resource Utilization

In this subsection, we present the FPGA resource utilization of the combined software-hardware drivers with different split points. We focus on the two main resources on FPGA, Look-Up Tables (LUTs) and Flip-Flops (FFs).

The drivers are implemented with Xilinx Vivado 2020.1 to the PL of ZynqMP Ultrascale+ MPSoC. The usage data is extracted from the Vivado report by including only the relevant components.

LUT and FF usages are shown in Figure 6.13 and Figure 6.14 respectively. Comparing the Xilinx IP with the generated driver at a similar abstraction

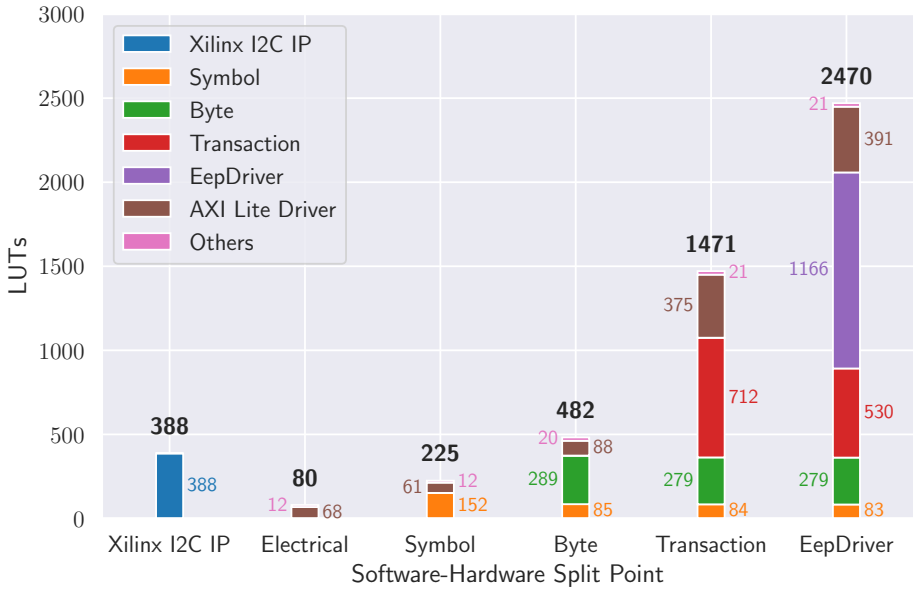


Figure 6.13: LUT usage with different software-hardware split points. The split point is the topmost layer in hardware, whose upper layer is the bottommost layer in software. "Others" is calculated as total minus the sum of the listed parts, which includes the I^2C clock divider and possibly other small parts.

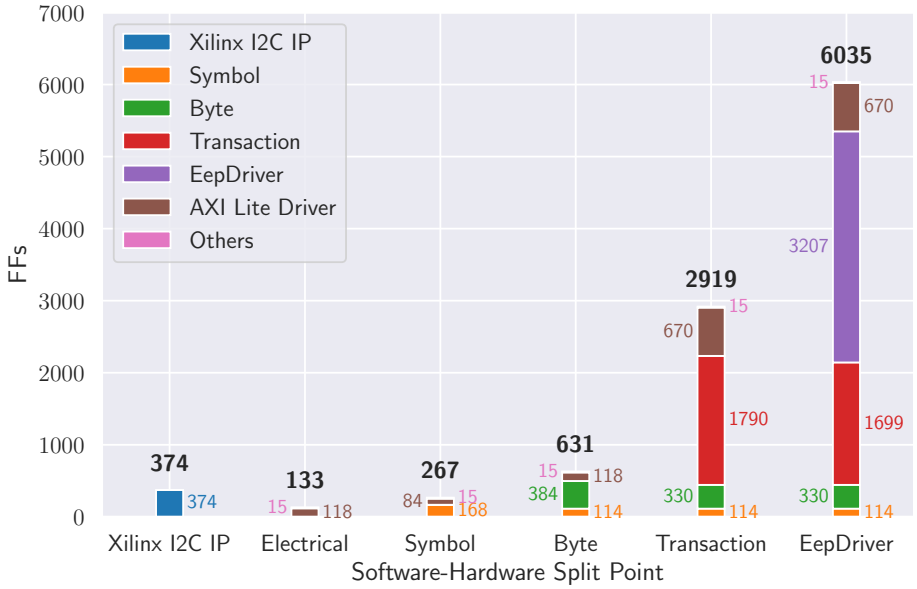


Figure 6.14: FF usage with different software-hardware split points. The split point is the topmost layer in hardware, whose upper layer is the bottommost layer in software. "Others" is calculated as total minus the sum of the listed parts, which includes the I^2C clock divider and possibly other small parts.

level—Transaction, the LUT usage of the generated driver is 4.35× and the FF usage is 7.80×.

In terms of absolute numbers, the generated drivers take much more resources compared to the Xilinx I^2C IP. Note that the Xilinx IP provides more sophisticated functionalities like interrupt and FIFO, so the comparison is already biased toward the generated drivers. In terms of the relative amount compared to the available resource on the FPGA (117120 LUTs, 234240 FFs), Xilinx IP takes 0.331% and 0.160% of the total available amounts respectively, while Transaction takes 1.26% and 1.25%, which are all small amounts.

There is a reasonable space for utilization optimizations in the generated drivers. For example, retaining the original specification, the current ESM specification still uses 32-bit integers for all data fields in the stack, even for 1-bit SCL and SDA signals and 8-bit byte payload. Even though Vivado is capable of optimizing unused registers away, exploring the possibilities remains as future work.

As a side note, as the split point moves up along the stack, we can see the resource usage of low layers decreases in Figure 6.13 and Figure 6.14. For example, the Symbol layer takes 152 LUTs when the split point is Symbol, but only takes 85 LUTs when the split point is Byte. We believe the underlying reason is Vivado performing cross-boundary optimization among components.

6.3.3 Effect of Optimizations

By choosing the LLVM IR as the input for the HDL backend, ESMC reuses the existing LLVM optimization passes. By default, O2 optimization is applied. In this subsection, we present the evaluation result on the effect of LLVM optimizations, as well as another optimization involved in the pipeline– the Xilinx Vivado FSM auto state encoding (discussed in Section 2.5). For completeness, we also present the resource usage of the hardware driver from Störzbach's work [27] when implemented on the current evaluation platform (different from the one reported in the work).

We use the EepDriver split point generated with the default configuration (LLVM O2, Vivado FSM optimization enabled) as the baseline. Subsequently, the LLVM optimization and the FSM optimization are disabled respectively. For Störzbach's driver, we take the code generated by the original compiler, plug it in the current FPGA design, and implement the design for the new BMC platform.

For timing, we do not observe significant differences between the default configurations and those with the optimization disabled. In terms of the FPGA resource utilization, Figure 6.15 and Figure 6.16 show the LUT and FF usages respectively. With LLVM optimization enabled, the LUT usage

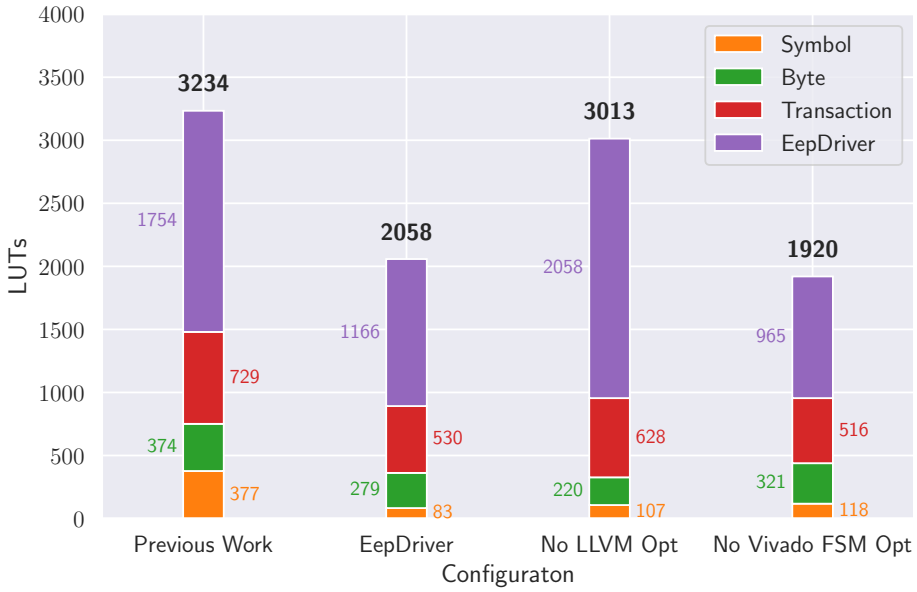


Figure 6.15: LUT usage with different configurations. Parts outside the I^2C stack are omitted for brevity .

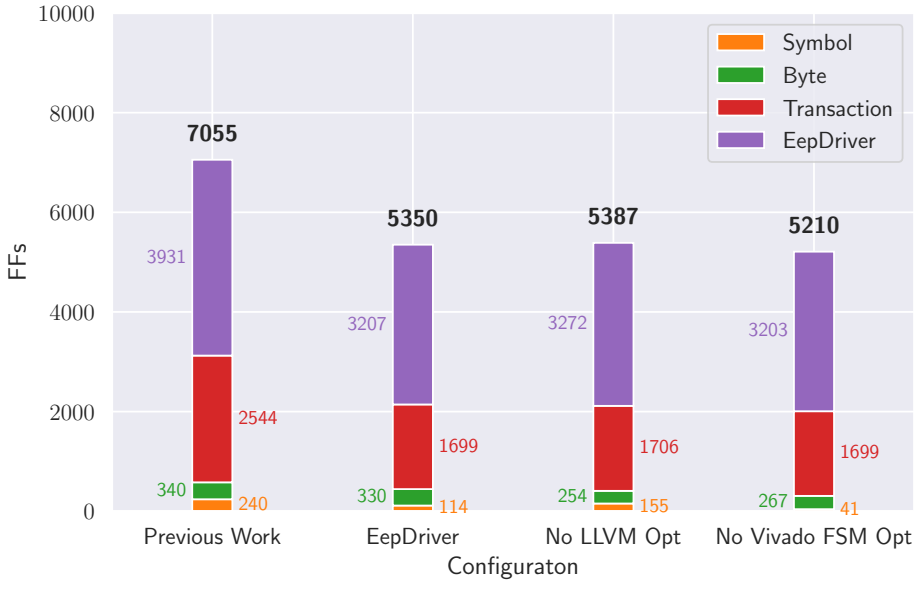


Figure 6.16: FF usage with different configurations. Parts outside the I^2C stack are omitted for brevity.

decreases by 31.5%. In contrast, FF usage decreases only by 0.687%. If using the previous work as the baseline, the generated driver reduces the LUT and FF usages by 36.4% and 24.2% of those the previous work. On the other hand, the Vivado FSM optimization increases both LUT and FF usage by a small amount.

Comparing EepDriver and No LLVM Opt, which varies only by if the LLVM optimizations are enabled, we can reach the conclusion that LLVM optimizations help reduce the resource usages. The effect on the LUT usage is significant, while the FF usage is less affected. On the other hand, the Vivado FSM optimization has a limited effect on the resource usage of the driver. The result is expected since that optimization is designed to optimize the state computing paths rather than the resource utilization.

When comparing with the previous work, we can see that the Efeu-based driver takes less resources. However, note that there are multiple variables that have changed: the compiler is completely changed, and the specification is slightly changed when migrating from the original DSL to ESM. Therefore, little conclusion can be be drawn from this observation.

There are multiple optimization passes in the LLVM O2 configuration. Isolating their individual effects remains a future work.

6.3.4 Crossing the Software-Hardware Boundary Multiple Times

So far, we focus on splitting the I^2C stack into two parts, with the high-level part implemented in software and the low-level part in hardware. While it is the typical design for combined software-hardware drivers, Efeu is far more flexible. As a demonstration, we design, implement, and evaluate a driver that cross the software-hardware boundary multiple times.

Figure 6.17 shows the architecture of the system. The top-level entry point (World layer) is in the software domain. The EepDriver lies in hardware. Subsequently, the Transaction layer is implemented again in software as another program. The remaining layers are again in hardware as another IP.

Implementing such a structure with the whole system compilation requires little work: the user simply configures multiple C/HDL components. The hierarchy of the output products by the ESMC is shown in Figure 6.18. For component 2, the layer has two MMIO-AXI Lite interfaces, one connected to the upstream and the other to the downstream, thereby there are two VHDL files describing the two interfaces. Similarly, for component 3, there are two C files describing its interfaces.

The components are compiled and deployed to the test platform described above. We validate the design with manual tests. Figure 6.19 shows the log. The output is expected, indicating the system works given the user inputs.

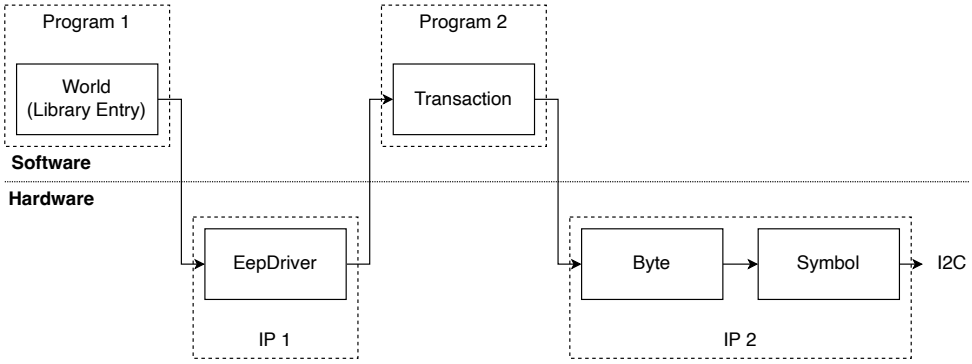


Figure 6.17: Architecture of a system crossing the software-hardware boundary multiple times. Boilerplate modules are omitted.

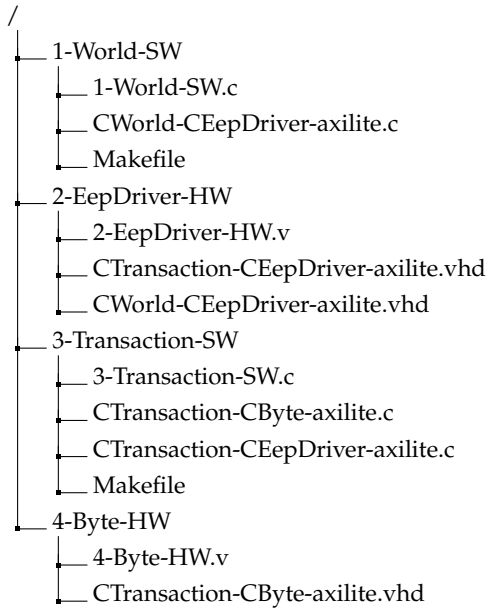


Figure 6.18: Directory hierarchy of the output product. Prefix "C" before layer names indicates "Controller".

```
1  root@mercury-xu5-1:~# ./snack-Transaction-SW &
2  [1] 316
3  [CTransaction] Start
4  root@mercury-xu5-1:~# ./snack-World-SW
5  Read  command: r <addr_in_eeprom> <length>
6  Write command: w <addr_in_eeprom> <data0> <data1> ...
7  > r 0x06 1
8  [CWorld Init] res: ME_RES_IDLE
9  [CTransaction] TR_ACT_WRITE, addr = 50, length = 2
10 [CTransaction] TR_ACT_READ, addr = 50, length = 1
11 [CTransaction] TR_ACT_STOP, addr = 0, length = 0
12 [CWorld] res: ME_RES_OK [0]1
13 > r 0x06 8
14 [CTransaction] TR_ACT_WRITE, addr = 50, length = 2
15 [CTransaction] TR_ACT_READ, addr = 50, length = 8
16 [CTransaction] TR_ACT_STOP, addr = 0, length = 0
17 [CWorld] res: ME_RES_OK [0]1 [1]2 [2]3 [3]4 [4]5 [5]6 [6]7 [7]8
18 > w 0x07 255 254 253 252
19 [CTransaction] TR_ACT_WRITE, addr = 50, length = 6
20 [CTransaction] TR_ACT_STOP, addr = 0, length = 0
21 [CWorld] res: ME_RES_OK
22 > r 0x08 8
23 [CTransaction] TR_ACT_WRITE, addr = 50, length = 2
24 [CTransaction] TR_ACT_READ, addr = 50, length = 8
25 [CTransaction] TR_ACT_STOP, addr = 0, length = 0
26 [CWorld] res: ME_RES_OK [0]254 [1]253 [2]252 [3]6 [4]7 [5]8 [6]9
       [7]10
```

Figure 6.19: Manual validation of the system. Lines starting with `[CTransaction]` ("C" indicates "Controller") are printed by program `snack-Transaction-SW`. Commands to Transaction start with `TR_ACT_`, followed by the target I^2C address (0x50 in this case) and length of the payload. Lines starting with `[CWorld]` are printed by program `snack-World-SW`. Lines starting with > are user input commands. `ME_RES_OK` is the return value from the top layer (CWorld), indicating the operation is done successfully. For read operations, returned data array follows, where the numbers in `[]` are indices into the data buffer.

Chapter 7

Discussion

7.1 Applications on BMC Firmware and Beyond

In this work, we demonstrate using Efeu to generate highly-assured and flexible I^2C device drivers. By design, Efeu is not limited to the I^2C stack, but applicable to systems that can be modeled as modules with synchronous communications in between. In addition to I^2C , other communication protocols used in embedded systems or baseboard management, can possibly benefit from Efeu, including Serial Peripheral Interface (SPI) [12], Controller Area Network (CAN) [30], Intelligent Platform Management Interface (IPMI) [17]. Similar to I^2C , these protocols define abstraction layers in their respective specifications, which possibly well align with the programming model of Efeu.

On the other hand, the I^2C stack can also be extended vertically. For instance, The System Management Bus (SMBus) [9] is a protocol built on I^2C , providing abstractions dedicated to system management. Enzian BMC uses SMBus to manage several components. To extend the stack to encompass SMBus, additional abstraction layers need to be modeled as Efeu layers. Looking ahead, one can envision extending the stack further upward to the BMC firmware control logic, such as the feedback control loop from CPU temperature to fan speeds.

While we employ a specific test platform in this work (Linux, ARMv8-based CPU, Xilinx FPGA), Efeu is designed with portability in mind. For example, to migrate the software drivers from Linux to seL4, only the boilerplate code like the MMIO driver needs to be adapted. The C code generated by Efeu is portable by nature and does not require any manual adaptation.

7.2 How Verifiers Help

When migrating the I²C stack specification from the original DSL to ESM, we leverage the existing verifiers as a tool for testing and debugging. In this section, we report one of the datapoints where the verifier reports errors, indicating problems in the driver. It serves as an example of how the verifiers help identify issues early in the development process.

In a version of controller EepDriver specification, a read action that should be issued to the downstream layer, Transaction, is missing (in the ESM code, `talkTransaction(TR_ACT_READ)`). In other words, the read request never reaches the Transaction layer but vanishes. The verifier for EepDriver reports "invalid end state (at depth 942)", indicating the Transaction layer keeps waiting. After fixing the bug, the error is resolved. This datapoint demonstrates how model checking helps prevent errors from going into production at an early stage of development.

7.3 Future Work

While Efeu provides a complete pipeline for designing and implementing highly assured drivers like I²C , it does come with certain limitations. In this section, we discuss the possible improvement to the current Efeu implementation as future work.

Formalization and Extension of the C Backend By leveraging C function calls and continuations, the current C backend provides a way to implement the `read` and `talk` operations with little overhead. However, the validity of this implementation has not been formalized. While the generated drivers work in our tests on real-world devices, the generated C code has a different semantics than the ESM specification: functions are not concurrent processes. Furthermore, there can possibly be systems that can be modeled with the semantics of concurrent processes, but not with C functions. Whether the two semantics are equivalent in limited conditions remains to be proved in future work.

Without diving into the formality, we briefly discuss a possible direction of such a proof. It is possible that the execution order of the C functions is *one of* the valid execution orders of the concurrent processes. In other words, with the function calls and continuations, the C backend implicitly performs a valid *static scheduling* on the processes. This property may only hold with some properties or conditions. For the I²C stack, passing the model checking ensures it is free of deadlocks and livelocks. If this is a necessary property to validate the C backend, the end user should be informed.

Last but not least, currently the C backend only supports calling trees–a layer can directly call multiple layers, but a reachable layer has one and only one caller. This constraint limits the C backend to be only capable of implementing a subset of systems allowed by the ESM specification. Implementing additional scheduling approaches, such as using threads to implement layers, remain as future work.

Interfaces with Dynamic Lengths Currently, Efeu only allows arrays with fixed sizes. With this limitation, users must declare arrays large enough to accommodate the maximum possible payload size, resulting in unnecessary data movement, latency, and resource usage. Currently, the I^2C stack supports payloads up to 16 bytes, which is sufficient for EEPROM. However, if the stack is extended to the SMBus protocol, the maximum payload size that needs to accommodate boosts to 256 bytes [9]. Implementing such an interface with the existing design is highly inefficient.

Features While The current Efeu programming model and the ESM language are sufficient for implementing the I^2C driver, there are yet several possible extensions to allow it to model more complicated systems. For instance, asynchronous communication mechanisms could be a useful feature. Channels with fixed-size queues are natively supported by Promela and have corresponding implementations in C and HDL. For ESM language, additional features can help improve usability for end users. For instance, while layer functions are the only allowed functions in ESM currently, helper functions can be allowed as long as they can be inlined by Clang/LLVM.

More Ways to `talk` Backends in Efeu essentially implement the `read` and `talk` operations in different ways: calls and continuations between layers in C, in-module handshaking between layers in HDL, and the MMIO AXI Lite interface for layers crossing the software and hardware boundary. In addition to these, one can envision more ways to `talk`. For example, there can be a way to `talk` between applications through inter-process communication mechanisms. Alternatively, there can be a way to `talk` between a VM and seL4 through some virtual device. With this backend, the user can isolate a part of the system in the VM, while implementing the trusted part as a native seL4 application.

Chapter 8

Conclusion

This work details Efeu, a framework for designing drivers for model checking and integrated software-hardware implementations. We demonstrate its usage on a model-checked I^2C stack. With the stack specified with Efeu's DSLs—ESM and ESI, Efeu can then translate the specification to various output products based on users' instructions. The Promela backend generates Promela specification of the stack, which can be used for model checking with the SPIN model checker. The C backend emits C code for software driver implementations. The HDL backend generates drivers in HDL that can be used to implement hardware drivers. Built upon these backends, the whole system compilation further enables the user to create combined software-hardware drivers, with the user freely choosing the split point between software and hardware components. The generated drivers are evaluated on real-world devices.

Looking forward, we envision extending Efeu with additional features, allowing it to model more complicated systems. We believe the success of the I^2C stack can be transferred to other communication protocols and other components built on top of I^2C , toward a highly-assured BMC firmware stack.

Appendix A

Complete ESI of the I²C Stack

The complete ESI of the I²C stack, including both the controller and the responder, is shown below. The "C" prefix indicates "Controller" and the "R" prefix indicates "Responder".

```
1  layer CElectrical;
2  layer CSymbol;
3  layer CByte;
4  layer CTransaction;
5  layer CEepDriver;
6  layer CWorld;
7
8  interface <CElectrical, CSymbol> {
9    => {
10     i32 scl;
11     i32 sda;
12   },
13   <= {
14     i32 scl;
15     i32 sda;
16   },
17 };
18
19 interface <CSymbol, CByte> {
20   => {
21     i32 sym;
22   },
23   <= {
24     i32 sym;
25   },
26 };
27
28 interface <CByte, CTransaction> {
29   => {
30     i32 res;
31     i32 data;
```

```
32    },
33    <= {
34      i32 action;
35      i32 data;
36    }
37  };
38
39  interface <CTransaction, CEepDriver> {
40    <= {
41      i32 action;
42      i32 addr;
43      i32 length;
44      i32 data[16];
45    },
46    => {
47      i32 res;
48      i32 length;
49      i32 data[16];
50    }
51  };
52
53  interface <CEepDriver, CWorld> {
54    <= {
55      i32 action;
56      i32 addr;
57      i32 length;
58      i32 data[16];
59    },
60    => {
61      i32 res;
62      i32 data[16];
63    }
64  };
65
66  layer RElectrical;
67  layer RSymbol;
68  layer RByte;
69  layer RTransaction;
70  layer REepDriver;
71  layer RWorld;
72
73  interface <RElectrical, RSymbol> {
74    => {
75      i32 scl;
76      i32 sda;
77    },
78    <= {
79      i32 scl;
80      i32 sda;
81    },
82  };
83
84  interface <RSymbol, RByte> {
85    => {
```

```
86      i32 sym;
87    },
88    <= {
89      i32 sym;
90    },
91  };
92
93  interface <RByte, RTransaction> {
94    => {
95      i32 res;
96      i32 data;
97    },
98    <= {
99      i32 action;
100     i32 data;
101   }
102 };
103
104 interface <RTransaction, REepDriver> {
105   <= {
106     i32 action;
107     i32 data;
108   },
109   => {
110     i32 res;
111     i32 data;
112   }
113 };
114
115 interface <REepDriver, RWorld> {
116   <= {
117     i32 action;
118     i32 length;
119     i32 data[16];
120   },
121   => {
122     i32 res;
123     i32 addr;
124     i32 length;
125     i32 data[16];
126   }
127 };
128
129 layer ElectricalBus2;
130
131 interface <CElectrical, ElectricalBus2> {
132   => {
133     i32 scl;
134     i32 sda;
135   },
136   <= {
137     i32 scl;
138     i32 sda;
139   },
```

```
140 };
141
142 interface <RElectrical, ElectricalBus2> {
143   => {
144     i32 scl;
145     i32 sda;
146   },
147   <= {
148     i32 scl;
149     i32 sda;
150   },
151 };
```

Appendix B

GPIO Bit-Banging Driver Through Linux sysfs

The GPIO bit-banging driver from the previous work [14, 15] is shown below. The code is reformatted for clarity. Lines with "MANUAL DELAY" are preserved when validating the functionality of the bit-banging software drivers in Section 6.1 and Subsection 6.2.1, but removed in Subsection 6.2.2.

```
1  #include <assert.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <time.h>
6  #include <unistd.h>
7  #define SDA_PIN 486
8  #define SCL_PIN 487
9
10 static void set_direction(int pin, const char *direction) {
11   FILE *f;
12   char fn[128];
13   snprintf(fn, sizeof(fn), "/sys/class/gpio/gpio%d/direction",
         pin);
14   f = fopen(fn, "w");
15   assert(f);
16   fprintf(f, "%s", direction);
17   fclose(f);
18 }
19
20 static int get_value(int pin) {
21   FILE *in;
22   char fn[128];
23   snprintf(fn, sizeof(fn), "/sys/class/gpio/gpio%d/value", pin);
24   in = fopen(fn, "r");
25   assert(in);
26   int val;
```

```
27    int conversions = fscanf(in, "%d", &val);
28    assert(conversions == 1);
29    fclose(in);
30    return val;
31  }
32
33  static void ms_sleep(int ms) { usleep(ms * 1000); }
34
35  static void i2c_set_pin(int pin, int val) {
36    // If val == 1 -> high impedance output
37    // If val == 0 -> drive low
38    if (val) {
39      set_direction(pin, "in");
40    } else {
41      set_direction(pin, "low");
42    }
43  }
44
45  void i2c_send(int scl, int sda) {
46    ms_sleep(2);  // MANUAL DELAY
47    if (scl) {
48      // Avoid generating spurious start/stop
49      i2c_set_pin(SDA_PIN, sda);
50      ms_sleep(2);  // MANUAL DELAY
51      i2c_set_pin(SCL_PIN, scl);
52    } else {
53      i2c_set_pin(SCL_PIN, scl);
54      ms_sleep(2);  // MANUAL DELAY
55      i2c_set_pin(SDA_PIN, sda);
56    }
57  }
58
59  void i2c_recv(int *scl, int *sda) {
60    ms_sleep(5);  // MANUAL DELAY
61    *scl = get_value(SCL_PIN);
62    *sda = get_value(SDA_PIN);
63  }
64
65  void i2c_init() {
66    FILE *f;
67
68    f = fopen("/sys/class/gpio/export", "w");
69    assert(f);
70    fprintf(f, "%d\n", SCL_PIN);
71    fclose(f);
72
73    f = fopen("/sys/class/gpio/export", "w");
74    assert(f);
75    fprintf(f, "%d\n", SDA_PIN);
76    fclose(f);
77
78    i2c_set_pin(SDA_PIN, 1);
79    i2c_set_pin(SCL_PIN, 1);
80  }
```

Acronyms

APU Application Processing Unit.

AST Abstract Syntax Tree.

BMC Baseboard Management Controller.

CPU Central Processing Unit.

DSL Domain-Specific Language.

EDA Electronic Design Automation.

EEPROM Electrically Erasable Programmable Read-Only Memory.

FF Flip-Flop.

FIFO First-In-First-Out.

FPGA Field Programmable Gate Array.

FSM Finite State Machine.

GPIO General-Purpose Input/Output.

HDL Hardware Description Languages.

HLS High-Level Synthesis.

IDE Integrated Development Environment.

IR Intermediate Representation.

LSB Least Significant Bit.

LUT Look-Up Table.

MMIO Memory-Mapped Input/Output.

MSB Most Significant Bit.

OS Operating System.

PL Programming Logic.

RAM Random-Access Memory.

SoC System-on-Chip.

SSA Static Single-Assignment.

VM Virtual Machine.

Bibliography

[1] Arm. *Learn the architecture - An introduction to AMBA AXI*.

[2] AustriaMicroSystems. *AS5011 Low power Integrated Hall IC for human interface applications*.

[3] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. Enzian: An open, general, cpu/fpga platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 434–451, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Christopher L Conway and Stephen A Edwards. NDL: A Domain-Specific Language for Device Drivers.

[5] The OpenBMC development community. Openbmc. `https://github.com/openbmc/openbmc`. Accessed: 2022-09-08.

[6] The SPIN development community. Promela Manual page. `http://spinroot.com/spin/Man/promela.html`. Accessed: 2023-08-02.

[7] Enclustra. *Mercury+ PE1 Base Board User Manual*.

[8] Enclustra. *Mercury XU5 SoC Module User Manual*.

[9] Roger Fan. SMBus Quick Start Guide. Technical Report 1, 2012.

[10] David Faragó and Peter H. Schmitt. Improving Non-Progress Cycle Checks. In Corina S. Păsăreanu, editor, *Model Checking Software*, volume 5578, pages 50–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[11] Ben Fiedler. Towards trustworthy bmc software on modern hardware. Master's thesis, ETH Zurich, 2021.

[12] Freescale Semiconductor. Using the Serial Peripheral Interface to Communicate Between Multiple Microcomputers. Technical Report 1, January 2002.

[13] Gernot Heiser. *The seL4 Microkernel: An Introduction*.

[14] Lukas Humbel. Modern hardware abstractions for firmware. Master's thesis, ETH Zurich, 2022.

[15] Lukas Humbel, Daniel Schwyn, Nora Hossle, Roni Haecki, Melissa Licciardello, Jan Schaer, David Cock, Michael Giardino, and Timothy Roscoe. A model-checked i2c specification. In Alfons Laarman and Ana Sokolova, editors, *Model Checking Software*, pages 177–193, Cham, 2021. Springer International Publishing.

[16] Giang Huong. Gcc2verilog compiler toolset for complete translation of c programming language into verilog hdl. *ETRI Journal*, 33:731–740, 10 2011.

[17] Intel, Hewlett-Packard, NEC, and Dell. Intelligent Platform Management Interface Specification Second Generation v2.0. Technical Report 2.0, October 2013.

[18] Keysight Technologies, Inc. Keysight InfiniiVision 3000T X-Series Oscilloscopes User's Guide, June 2020.

[19] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1), feb 2014.

[20] Microchip. *24AA512/24LC512/24FC512 512K I2C Serial EEPROM*.

[21] Stephen Neuendorffer. 2021 llvm dev mtg "architecting out-of-tree llvm projects using cmake". `https://www.youtube.com/watch?v=7wOU7csj1ME`. Accessed: 2023-04-26.

[22] M. O'Nils, J. Oberg, and A. Jantsch. Grammar based modelling and synthesis of device drivers and bus interfaces. In *Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204)*, volume 1, pages 55–58 vol.1, August 1998. ISSN: 1089-6503.

[23] Fabrice Merillon Laurent Reveillere, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for Hardware Programming.

[24] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 73–86, Big Sky Montana USA, October 2009. ACM.

[25] Jasmin Schult, Daniel Schwyn, Michael Giardino, David Cock, Reto Achermann, and Timothy Roscoe. Declarative Power Sequencing. *ACM Transactions on Embedded Computing Systems*, 20(5s):1–21, October 2021.

[26] NXP Semiconductors. *I2C-bus specification and user manual*.

[27] Pascal Störzbach. Generating vhdl implementation from formal description of i2c protocol. Master's thesis, ETH Zurich, 2022.

[28] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam. HAIL: a language for easy and correct device access. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 1–9, Jersey City NJ USA, September 2005. ACM.

[29] The Clang Team. Assembling a complete toolchain. `https://clang.llvm.org/docs/Toolchain.html`. Accessed: 2023-08-29.

[30] Texas Instruments. Introduction to the Controller Area Network (CAN). Technical report, May 2016.

[31] Shaojie Wang, S. Malik, and R.A. Bergamaschi. Modeling and integration of peripheral devices in embedded systems. In *Automation and Test in Europe Conference and Exhibition 2003 Design*, pages 136–141, March 2003. ISSN: 1530-1591.

[32] Xilinx. *Zynq UltraScale+ Device Technical Reference Manual*.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Generating Trustworthy I2C Stacks Across Software and Hardware

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Liu	Zikai

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich 17.09.2023	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.