ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 342

Systems Group, Department of Computer Science, ETH Zurich

Towards Trustworthy BMC Software on Modern Hardware

by

Ben Fiedler

Supervised by

Prof. Timothy Roscoe
Dr. David Cock
Dr. Michael Giardino

February 2021–August 2021

D INFK

Abstract

Board management controllers (BMCs) are ubiquitous in today's computers. They run in parallel to the computer's operating system and fulfil tasks such as hardware initialisation, system monitoring and provide low-level remote access to the system using web interfaces or console redirection. Such low-level access to a system means that BMCs must be *trusted*, however, in practice, implementations are seldom *trustworthy*. BMCs commonly ship as opaque binary blobs with the hardware they run on, forcing users to run unknown, untrusted, highly privileged code. Open-source projects such as OpenBMC and u-bmc have been developed in recent years, claiming to be safe(r) alternatives to the status quo, and show that there is interest in trustworthy BMCs. We take a different approach: hardware fault handling is important to guarantee safe system operation. We develop a formally verified, abstract fault handler model and security properties to ensure correct operation. Finally, we implement a concrete fault handler for the Enzian platform and show that is it faithful with respect to our model, even considering misbehaving devices. All proofs are formally verified in Isabelle/HOL. We prove that writing formally verified BMC software is not only feasible, but also possible, and take the first step towards a fully verified BMC for the Enzian platform.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Program testing can be used to show the presence of bugs, but never to show their absence!

— Edsger W. Dijkstra, Notes on Structured Programming

Modern computers are complex machines. It does not matter if we talk about about embedded microcontrollers or rack-scale servers: they all consist of many moving parts, both on the software and hardware level.

The baseboard management controller (BMC) is one of the most trusted, yet least scrutinised components of modern computers. It is a general-purpose computer present in essentially every system, which is responsible for tasks such as power sequencing, thermal monitoring and management, fault reporting, console redirection and much more.

The benefits of introducing such complexity at a low level in a system are clear: complex functionality can be implemented in software rather than hardware, which is easier to update in case of new processor revisions or bugs in the firmware. Additionally, they provide remote management capabilities, which are very useful for dealing with hardware located in datacenters. However, the downsides are also significant: such low level access to the system means the BMC must be completely trusted.

BMCs generally run a general-purpose operating system, such as Minix or Linux, with critical functionality commonly implemented using low-assurance techniques. Worse yet, many BMC implementations are network-accessible and thus remotely exploitable by malicious actors. Many vulnerabilities have already been published[1]. Almost all BMC implementations these days are

[1] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=BMC

closed-source software, distributed as opaque binary images that run on the server board, which additionally requires trust in the board manufacturer.

Thus a conundrum emerges: BMCs are necessarily *trusted*, but not *trustworthy*. The OpenBMC project [28], initially started by IBM, provides an open-source fully-fledged BMC implementation. It removes the necessity to place trust in the board manufacturer, since the code that runs on the BMC is publicly available. Still, OpenBMC is a full Linux distribution, using systemd [29] as a service manager. These tools, while well-tested, are not infallible.

We believe that a formally verified, open source implementation is the best way to ensure that the low-level BMC is both trustworthy and secure. Our first step towards this long-term goal is modelling and verifying one of the most important components of a BMC: the fault handling mechanism. Faults, which we also call alerts, are generated by on-board components when their operating characteristics approach their electrical or thermal limits. It is crucial to the system safety that a BMC correctly and quickly deals with these situations, otherwise the underlying hardware could be damaged severely.

In this thesis we develop an abstract alert handling model, resulting from generalising a concrete alert handling implementation for integrated circuits attached to the System Management Bus (SMBus) present in modern hardware systems, namely the devices on board of the Enzian[2] platform. We identify two necessary safety and liveness properties that every alert handler should satisfy: (1) all received alerts are handled, and (2) critical alerts eventually lead to a platform halt. Our model and proofs are formalised in Isabelle/HOL [23].

We build upon previous work by Heimhofer [8], who developed a minimal working seL4-based [14] BMC for the Enzian platform, using the CAmkES [17] component architecture. Our main contribution is a verified alert handling implementation for the Enzian BMC written in C. We use state-of-the-art tools [4] to convert our C implementation to a verifiable representation in Isabelle/HOL. Using Hoare logic [10], we formally describe and prove the BMC's alert handling behaviour, which we then relate to our abstract model. Due to upstream tooling still being developed,

The concrete alert handling implementation for SMBus devices we developed is tied to the Enzian platform, however our abstract model is not. Our model is flexible enough to survive contact with the real-world, such as misbehaving devices or faulty hardware.

[2] http://enzian.systems

1.1 Related work

The seL4 microkernel and CAmkES have been used for building high-assurance systems. Not only do they offer isolation proofs between different components, but they also allow execution of regular Linux VMs, which is helpful for porting existing, low-assurance systems to seL4/CAmkES [15], as it allows one to decompose a monolithic system step-by-step into verified parts.

Cyber-physical systems are a related area which relies on formal methods for guaranteeing secure operation. Failure to uphold the safety guarantees in cyber-physical systems may have severe, if not fatal, consequences, such as failed orbital launches [18]. VeriPhy [1] is an example of a full-stack verification toolchain for cyber-physical systems, using domain-specific modelling tools [5] as well as Isabelle/HOL, and CakeML [16].

Research into BMC implementations and verification is much more sparse. Most BMC implementations provided by hardware vendors are proprietary and closed-source. OpenBMC [28] and u-bmc [2] are two open-source BMC projects. While these implementations do not use high-assurance techniques such as formal verification, they represent a shift in the general attitude towards highly privileged low-level system components: both projects advertise themselves as safe(r) than the status quo of vendor-specific firmware.

Chapter 2

Background

Bei einem Dichter klauen ist Diebstahl, bei vielen Dichtern klauen ist Recherche.

— Walter Moers

In this chapter we present a brief overview of the technologies and software we used. Our work is not clearly situated in a single subarea of computer science research, but touches upon logic, formal methods and verification, operating systems research and hardware management.

2.1 seL4 and CAmkES

The seL4 microkernel [14] is a high-assurance, high-performance operating system microkernel. It belongs to the L4 microkernel family [20], and is the world's first fully verified microkernel. Initially targeting the ARM architecture, it has since expanded to cover the x86, x86-64 and RISC-V architectures as well.

CAmkES [17] is short for *component architecture for microkernel-based embedded systems*. It offers a framework for building and composing microkernel-based operating systems. It follows a component-based software engineering approach, modelling systems as set of components, interacting via explicit and well-defined interfaces.

Although CAmkES is not tied to any concrete microkernel implementation, seL4 provides full integration with CAmkES out of the box. Each component is run in its own address space and communication is done via synchronous and asynchronous message passing.

2.2 Isabelle/HOL

Isabelle/HOL [23] is a proof assistant based on classical higher-order logic. Its syntax is strongly influenced by the functional programming language Standard ML [22], with extensions for universal and existential quantifiers, sets and other common mathematical notation.

As in Standard ML, type variables are denoted by `'a`, `'b`. Common datatypes such as `bool`, `nat`, `int` are predefined in HOL, as are fundamental datatypes such as sets and lists, which are denoted using postfix notation, e.g. `nat list` denotes a list of natural numbers, and `'a set list` a list of sets of `'a`s. Tuples are denoted `(a, b)` and have type `a * b`.

Listing 2.1 shows an example of defining the natural numbers `N` with addition in Isabelle/HOL. The lemma `one_plus_one` proves its goal by repeatedly applying the rewrite rules associated with the `plus` function.

$$\texttt{plus (S Z) (S Z)} \xrightarrow{\texttt{plus.2}} \texttt{S (plus (S Z) Z)} \xrightarrow{\texttt{plus.1}} \texttt{S (S Z)}$$

There are two common styles used for writing formal proofs with Isabelle: the older apply-style proofs and the newer Isar proofs. Isar proofs are structured and represent each proof step explicitly, while the apply-style proofs represent the proof state implicitly. The two `plus_Z` lemmas in Listing 2.1 illustrate both proof methods. In general, structured Isar proofs are preferred to apply-style proofs. Exceptions can be made if specifying the intermediate proof state is unwieldy or unneccesary. They also show how powerful HOL is compared to Standard ML: the `datatype` definition of the natural numbers automatically derived a sound induction rule for us, without us having to expend any additional work!

```
datatype N = Z | S N

fun plus :: "N ⇒ N ⇒ N" where
  "plus n Z = n"
| "plus n (S m) = S (plus n m)"

lemma one_plus_one:
  "plus (S Z) (S Z) = S (S Z)"
  by simp
```

```
— Isar proof
lemma plus_Z:
  "plus Z m = m"
proof (induct m)
  case Z
  then show ?case by simp
next
  case (S m)
  then show ?case by auto
qed
```

```
— apply-style proof
lemma plus_Z':
  "plus Z m = m"
  apply (induct m)
   apply auto
  done
```

Listing 2.1: Examples of Isabelle/HOL definitions and proofs

Isabelle features a variety of proof tactics such as term rewriting, tableaux provers and term resolution. Additionally, it can delegate proof goals to a number of SMT solvers, whose proofs are cross-checked against Isabelle's own kernel, enabling powerful proof automation without sacrificing soundness. This form of proof automation allows users to focus on high-level goals, while the system deals with technicalities by itself. Table 2.1 denotes common proof tactics used in Isabelle proofs and which concepts they are based upon.

Proof tactic	**Mathematical concepts**
`simp`	Repeatedly applies rewrite rules until no more matches are found. Always terminates unless the user adds rules that lead to a rewriting loop.
`auto`	More aggressive than `simp`, can solve more complicated problems but does not guarantee termination
`force`, `fastforce`	Similar to `auto`, try harder to solve subgoals before timing out
`blast`	Tableaux prover using first-order unification, works well on first-order logic and set theory
`metis`	Generic resolution prover
`sledgehammer`	Meta-prover that invokes external SMT solvers and replays the proof using Isabelle's trusted kernel

Table 2.1: Common Isabelle proof tactics

Additionally, we make use of an Isabelle feature known as *locales*, which allows us to fix constants and variables with some assumptions about them without re-stating these assumptions in every lemma. A simple example of a locale defining algebraic groups is shown in Listing 2.2.

If we provide an *interpretation* of the locale's constants such that the assumptions hold, then we can also derive to all facts proved about the abstract locale. For example, the integers form a group under addition with identity 0, which we can easily prove, as shown in Listing 2.3.

Isabelle/HOL has a large standard library containing many algorithms, datastructures and locales with their corresponding correctness proofs.

```
locale group =
  fixes
    id :: 'a and
    op :: "'a ⇒ 'a ⇒ 'a" (infixr "⊕" 60) and
    inv :: "'a ⇒ 'a"
  assumes
    assoc: "(a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)" and
    ident_l: "a ⊕ id = a" and
    inv_r: "a ⊕ (inv a) = id"
begin

lemma inv_l:
  "(inv a) ⊕ a = id"
  by (metis assoc ident_l inv_r)

lemma ident_r:
  "id ⊕ a = a"
  by (metis assoc ident_l inv_r)

end
```

Listing 2.2: Abstractly defining algebraic groups via locales in Isabelle/HOL

```
interpretation i: group
  "0 :: int"
  "(+) :: int ⇒ int ⇒ int"
  "(λi. -i) :: int ⇒ int"
proof (unfold_locales)
  show "⋀a b c::int. (a + b) + c = a + (b + c)"
    by simp
  show "⋀a::int. a + 0 = a"
    by force
  show "⋀a::int. a + (- a) = 0"
    by fastforce
qed

— only proven within the locale!
— prints 0 + ?a = ?a
thm i.ident_r
```

Listing 2.3: Interpreting addition over the integers an algebraic group

2.3 Simpl and AutoCorres

The Simpl language is a sequential, imperative programming language [25], whose syntax and semantics (both big-step and small-step) are formalized in Isabelle/HOL. The Simpl language is flexible, and can be used as a model for representing concrete programs in an abstract, well-defined context, which is especially useful in program verification. Its syntax is similar to pseudocode, as seen in this sample program:

```
1 DO
2    `N :== `N - 1;;
3    IF `N = 0 THEN
4      `M :== 2 * `M
5    ELSE
6      `M :== `M + 1
7    FI
8 OD
```

Schirmer defines semantics of two Hoare logics [10] for Simpl, which can be used to reason about behaviour of Simpl programs. Hoare logics allow us to express properties of (parts of) a program using *pre-* and *postconditions*. A Hoare triple $\{\!|P|\!\}\ f\ \{\!|Q|\!\}$, requires that if P holds before f, then Q holds after execution of f.

Partial correctness does not require f to terminate, whereas *total correctness* requires termination of f. Simpl includes a verification condition generator (VCG) tactic which aids in reasoning about the behaviour of Simpl programs. An example of a Simpl statement and the VCG tactic is shown in Listing 2.4. It transforms the statement about Simpl code in a logical implication, which can be proven with conventional Isabelle tools. Using Isabelle's custom notation support, the resulting strucure looks similar to conventional notation in Hoare logic.

Schirmer's work also includes the definition of a type-safe subset of C99 [13], denoted C0, and an embedding of the C0 language in Simpl, including a

```
lemma simpl_vcg_example:
  "Γ ⊢ {| ´N > 0 ∧ ´M < 0 |}
       DO
         ´N :== ´N - 1;;
         ´M :== 2 * ´M
       OD
       {| ´N ≥ 0 ∧ ´M < 0 |}"
  apply vcg
  — have to show ∀N M. 0 < N ∧ M < 0 ⟶ 0 ≤ N - 1 ∧ 2 * M < 0
  apply simp
  done
```

Listing 2.4: Example of Simpl's Hoare Logic and VCG

```
type_synonym ('s, 'a) nondet_monad = "'s ⇒ ('a × 's) set × bool"

definition
  return :: "'a ⇒ ('s, 'a) nondet_monad" where
 "return a ≡ λs. ({(a,s)},False)"

definition
  bind :: "('s, 'a) nondet_monad ⇒ ('a ⇒ ('s, 'b) nondet_monad) ⇒
           ('s, 'b) nondet_monad" (infixl ">>=" 60) where
 "bind f g ≡ λs. (⋃(fst ` case_prod g ` fst (f s)),
                  True ∈ snd ` case_prod g ` fst (f s) ∨ snd (f s))"
```

Listing 2.5: Definition of `nondet_monad` according to Cock et al. [3]

correspondence proof which allows properties proven about the Simpl representation to be applied to the C0 program [25, Chapters 6 and 7]. However, Schirmer's work relies on the user manually translating C0 programs into valid C0 ASTs, represented as Isabelle/HOL datatypes.

The Data61 group extended Schirmer's work and developed a parser[1] [4], which automates translation of C to its C0 AST form. It is used, among others, as part the seL4 correctness proof, parsing and transforming the kernel source code into a Simpl representation. The C-to-Simpl translator is conservative in its operation and aims to translate the underlying code faithfully, exposing low-level implementation details of the original C program to the Simpl layer.

AutoCorres [6] was developed to abstract the Simpl representation of C code even further, making the resulting structure easier for humans to reason about. It transforms Simpl programs to a monadic representation, first described by Cock et al. [3], and automatically proves correspondence of the abstracted program and the concrete Simpl implementation.

Cock et al. define a state monad `('s, 'a) nondet_monad` in Isabelle/HOL, and use it to express computations which may (1) mutate global state `'s`, (2) produce nondeterministic (including empty) results and (3) fail. The type `'s` denotes the state type, while the type `'a` denotes the result type of the monad. Listing 2.5 shows the Isabelle/HOL definition of `nondet_monad`. Even though the definition seems complicated at first glance, the semantics of `return` and `bind` are straightforward: `return` wraps a value of type `'a`, and `bind` combines the sequential execution of two monads `f` and `g`: First `f` is run, and then `g` is run on all resulting states of `f`, while taking care that the failure flag is propagated correctly.

State is used to represent the global state of the C program. Nondeterminism naturally arises in many cases, such as interaction with hardware components

[1] The C-to-Simpl parser generates Simpl code from the C code, so it might better be described as *translator* instead of *parser*. However tool itself is named *c-parser*, so we use both terms interchangeably.

```
definition valid ("⦃_⦄/ _ /⦃_⦄") where
  "valid P f Q ≡
     ∀s. P s ⟶ (∀(r,s') ∈ fst (f s). Q r s')"

definition no_fail where
  "no_fail P m ≡ ∀s. P s ⟶ ¬ (snd (m s))"

definition validNF where
  "validNF P f Q ≡ valid P f Q ∧ no_fail P f"
```

Listing 2.6: Partial and total correctness definitions for `nondet_monad`

as we see in Chapter 4. Finally, failure is part of the C (and thus C0) semantics, as is the case with out-of-bounds array accesses and null pointer dereferences for example.

These concepts are important to distinguish the two different Hoare Logics formalised for `nondet_monad`: For a computation `f` and conditions P and Q, partial correctness ignores failure and nontermination, while total correctness requires `f` to be failure-free (and also terminating), similar to the Simpl definitions. For monadic programs, we denote total correctness by adding an exclamation mark to the formula: $\{\!|P|\!\}$ `f` $\{\!|Q|\!\}!$. These definitions are shown in Listing 2.6.

Note that total correctness of a monadic program specification `f` does not imply executability: if no result state for `f` exists then the total correctness Hoare triple $\{\!|P|\!\}$ `f` $\{\!|Q|\!\}!$ holds trivially.

To simplify proofs about the behaviours of functions, Cock et al. [3] also developed a verification condition generator based on Hoare rules in weakest-precondition form. An example is shown in Listing 2.7. The VCG can be extended by the user with additional rules, which allows them to build higher and higher abstractions of their model. Furthermore, these rules do not strictly have to be in weakest-precondition form: the soundness of the VCG is not affected by them, however it risks generating unprovable goals, indicating that the supplied rules have to be strengthened.

```
lemma monadic_vcg_example:
  "{| λ(n::nat, m::int). n > 0 ∧ m < 0 |}
     do
        modify (λ(n, m). (n-1, m));
        modify (λ(n, m). (n, 2*m))
     od
   {| λ_ (n, m). n ≥ 0 ∧ m < 0 |}"
  apply wp
  — have to show ∀n m. 0 < n ∧ m < 0 ⟶ 0 ≤ n - 1 ∧ 2 * m < 0
  apply auto
  done
```

Listing 2.7: Example of AutoCorres' `wp` tactic

2.4 I²C, SMBus and PMBus

The inter-integrated circuit (I²C) bus [27] was developed in 1982 by Philips Semiconductors as lightweight, serial communication bus between integrated circuits (ICs). It is a fundamental building block of modern computing systems, used in almost all modern computing hardware, from mobile phones to server platforms.

I²C specifies two open drain lines, a data line (SDA) and clock line (SDC). Devices on the bus are addressed by 7-bit identifiers, 16 of which are reserved by the I²C specification. I²C buses can have multiple controllers and multiple peripherals. I²C data is transferred in *transactions*. Each transaction is delimited by start and stop symbols and may contain multiple *messages* of an arbirary number of bits. We will not go into detail about the electrical details of the I²C specification, as it is irrelevant to this thesis.

I²C data transfer is not standardised beyond the bit level, and as such each device has to define its own data transfer protocol. The System Management Bus (SMBus) was later specified as a restriction of the I²C bus, defining a structured data interchange format between ICs. Additionally, SMBus introduces 256 *registers*[2], which are an 8-bit identifier used to distinguish the semantic of a read or write issued to a peripheral. Examples of the structured data exchange commands are the read-byte or write-word commands. SMBus is electrically compatible with I²C bus.

The `SMBALERT` line is an optinal SMBus feature, which is used by peripherals to notify the bus controller of a pending interrupt. Additionally, it specifies the Alert Response Address (ARA) protocol, whose purpose is to allow the controller to determine who is holding the `SMBALERT` line. Issuing a read from a special I²C address, the Alert Response Address, returns an alerting device's address[3]. Peripherals are required to release the `SMBALERT` line immediately

[2] the official specification calls these *command codes*

[3] Specifically the device with the *lowest* address, due to the I²C electrical characteristics.

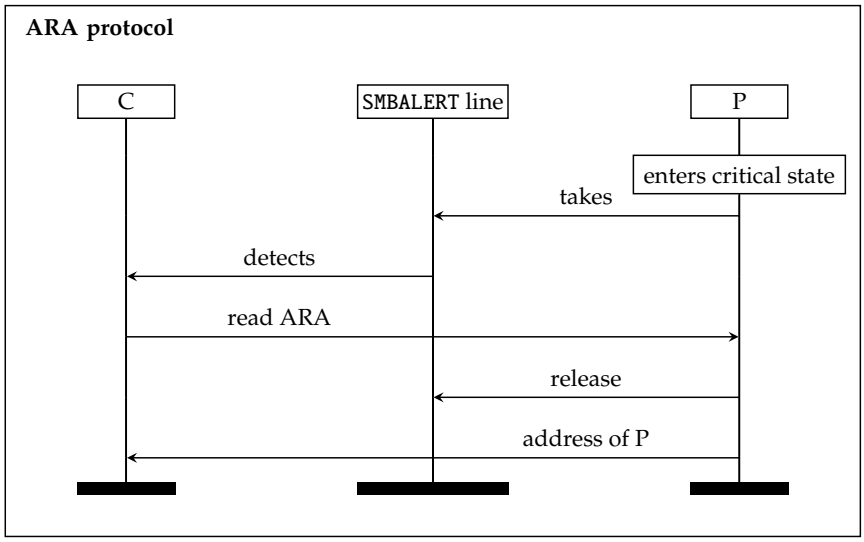

Figure 2.1: Example of the ARA protocol in action

once they have responded to the ARA request. Thus, if the alert line is still held after completing an ARA request, the controller can infer that there is at least one additional alerting device. An example of a controller C handling a device alert from a peripheral P is shown in Figure 2.1.

Although SMBus improved upon the I^2C specification by defining a structured data interchange format, the command semantics varied greatly between different ICs. The Power Management Bus (PMBus) specification extends the SMBus specification by defining semantics for a number of common commands across many devices. An example is the `OT_FAULT_LIMIT` command, which sets the minimum temperature at which the device will report an overtemperature fault. PMBus is electrically compatible with SMBus.

PMBus devices also use the `SMBALERT` line for alerting the host controller. Furthermore, PMBus defines status query commands `STATUS_BYTE` and `STATUS_WORD`, which describe the alert state of various monitors on the peripheral: temperature sensors, voltage monitors, current monitors and more.

Since all three standards are electrically compatible, it is common that I^2C, SMBus and PMBus devices share the same bus.

2.5 Linear Temporal Logic

Linear Temporal Logic (LTL), first described in 1977 by Pnueli [24], is a formal logic for expressing the behaviour of infinite sequences of truth valuation some finite ground set of variables V. LTL formulas are defined inductively as follows:

1. If $a \in V$, then a is an LTL formula.
2. If φ and ψ are LTL formulas, then $\neg\varphi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, $\circ\varphi$ and $\varphi \,\mathcal{U}\, \psi$ are an LTL formulas.

The logical connectives $\neg$ and $\vee$ are complete for predicate logic, which means we can derive the standard logical operators $\wedge$, $\rightarrow$, and so on as well. The two LTL-specific operators are $\circ$ and $\mathcal{U}$, whose semantics are defined as follows.

Let s be a sequence of truth valuations of the variables V. We define s_i to denote the i-th valuation of s, starting at 0. The formula $\circ\varphi$ is true if and only if, for a stream s, φ holds in the *next* step

$$\circ\varphi \leftrightarrow \varphi \text{ holds in } s_1$$

The operator $\mathcal{U}$ is called the *until* operator. The formula $\varphi \,\mathcal{U}\, \psi$ is true if φ always holds until ψ holds. Formally,

$$\varphi \,\mathcal{U}\, \psi \leftrightarrow \exists i \in \mathbb{N}_0.\ \psi \text{ holds in } s_i \wedge \forall j < i.\ \varphi \text{ holds in } s_j$$

Note that choosing $i = 0$ is valid if ψ holds immediately.

Using these LTL primitives, one can derive additional operators, which are displayed in Table 2.2. Ultimately, we are interested in formally verifying LTL formulas, and fortunately for us the Isabelle system distribution ships with a LTL library that formally defines all operators we need.

LTL formula	**Isabelle**	**Semantic**
$\circ\varphi$	`next` φ	φ holds in the next step
$\Diamond\varphi$	`ev` φ	φ eventually holds
$\Box\varphi$	`alw` φ	φ always holds
$\varphi \,\mathcal{U}\, \psi$	φ `until` ψ	φ holds until ψ holds

Table 2.2: LTL operators

There are many more LTL operators that can be derived, however these are all operators needed to understand the LTL formulas in this thesis.

Chapter 3

Abstract characterization of alert handling

Do you wish me a good morning, or mean that it is a good morning whether I want it or not; or that you feel good this morning; or that it is a morning to be good on?

— Gandalf to Bilbo in *The Hobbit* by J. R. R. Tolkien

Before we can implement a alert handler at all, we need to define the semantics of alert handling. Starting from a very concrete alert handling example, we develop a general alert handling model, and show that our model satisfies two important security properties: (1) every alert gets handled and (2) critical alert lead to a platform shutdown. Proving property (2) depends on *weak fairness* gurantees for some actions in the model, which have to be proven separately.

3.1 Abstracting the SMBus alerting procedure

Figure 3.1 shows the SMBus alerting model for a bus controller C and a peripheral P. In our concrete case, the Enzian BMC fulfils the role of the controller C for many, potentially different, peripherals P. Once an alert occurs on some peripheral P, the BMC has to run the ARA protocol (explained in Section 2.4) to determine the alerting device. Then, the BMCs fetches detailed alert information and decides what should happen next. If, for example, the alert is a critical overtemperature fault, then the next step should be to shut down the platform to prevent damage to the hardware.

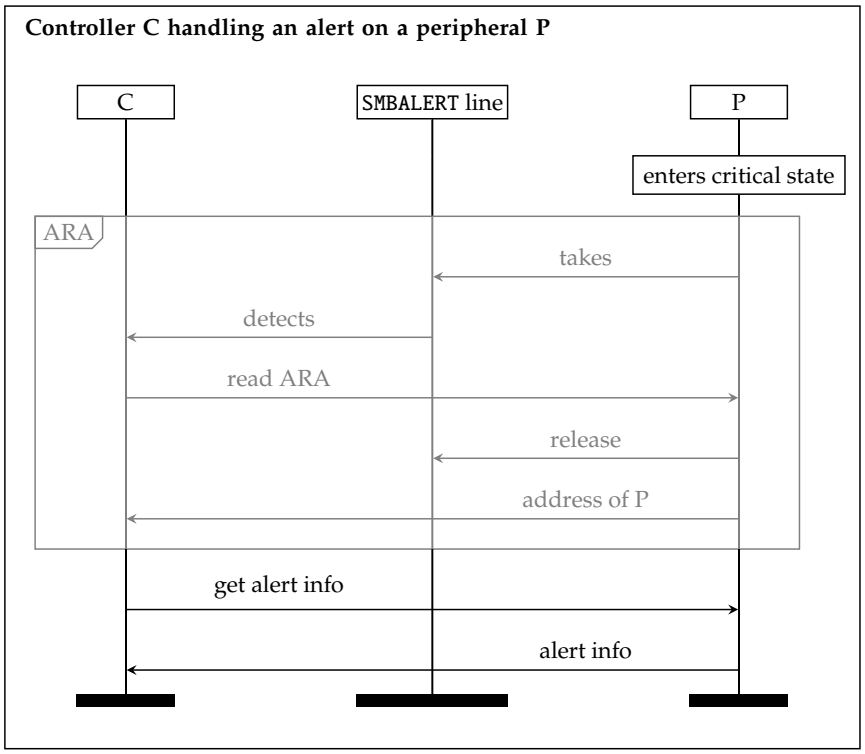

Figure 3.1: Full alert receiving procedure for an SMBus alert on the Enzian BMC

Algorithm 1: Alert handling procedure for a controller of a single SMBus

```
1  while alert line held low do
2  |  addr ← get alerting device's address;
3  |  d ← nil;
4  |  if device at address addr has additional alert details then
5  |  |  d ← query alert details;
6  |  end
7  |  critical ← decide whether (addr, d) is a critical alert;
8  |  if critical then
9  |  |  shutdown platform;
10 |  end
11 end
```

In general, each peripheral P connected to an SMBus is a potential *alert source*, while the BMC acts as an *alert receiver*, responsible for collecting alert details and correctly handling alerts. Thus, we can partition the resposibilities of the BMC as an alert handler as follows:

1. Acquiring alert details (address of alerting device, alert kind, etc.)
2. Consuming all received alerts and deciding if a shutdown is needed
3. (if necessary) Shut down the platform

Naïvely translating these steps into an algorithm results in something similar to Algorithm 1. Algorithm 1 is correct with respect to the semantics of the SMBus alerting schema. However, this code is not particularly suited for verification: defining semantics for Algorithm 1 is difficult because it combines multiple device and alert handler state transitions in one function. Reasoning about transient intermediate states is cumbersome and error prone.

Instead, we implement the three alert handling tasks separately: each part has a well-defined, limited reponsibility and allows us to assign clear semantics to it. Alerts are initially received from a source by an alert receiver, and then persisted until they are consumed.

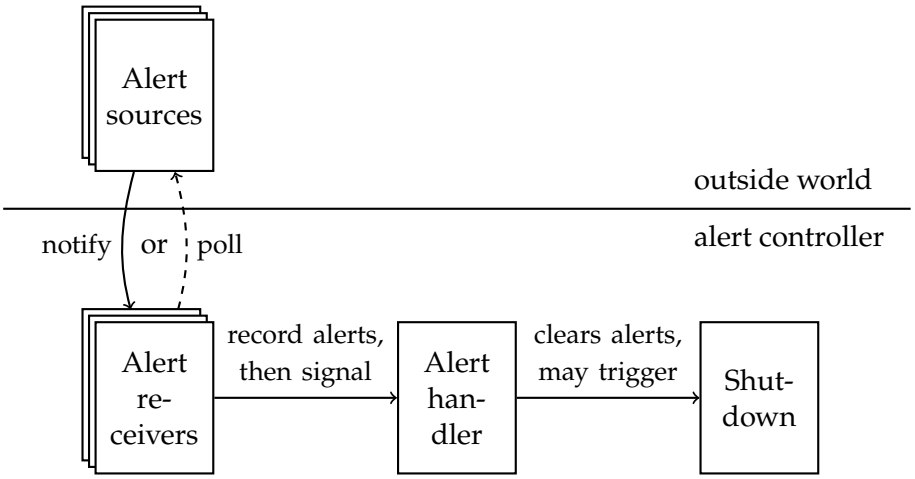

Figure 3.2: Alert handling model abstracted from Figure 3.1

We limit the number of alerts stored to some arbitrary but fixed natural number n. An abstract model does not require this in any way, however all its implementations are constrained by a finite amount of memory, and hence can only store a finite number of alerts. As long as strictly less than n alerts have been observed, the controller has to correctly record incoming alerts. No guarantees apply to any alerts received beyond that.

Additionally, we allow each alert receiver to mask all incoming alerts. If the alerts for a receiver are masked, then the controller must not change its state

when receiving an alert on that receiver; otherwise it proceeds with the flow devised in Figure 3.2.

3.2 Isabelle model

Having informally derived our alert handling model in the previous section, we formalise it in Isabelle/HOL. The alert controller's state is parametrised by two types: the *receiver type* `'r`, which ranges over all possible alert receivers, on, and the *alert details* `'a`, which represents information relevant to an alert. The exact semantics of these types are to be determined by the concrete implementation.

The state itself is composed of three parts: information about the masked state of each alert receiver, the (multi-)set of alerts that have been received, but not handled yet, and a boolean to indicate whether the platform should shut down. The controller's behaviour is characterised by an infinite stream of states, which we call a *trace*. The state definition is shown in Listing 3.1.

```
record ('r, 'a) state =
  alerts_enabled :: "'r ⇒ bool"
  alerts :: "'a multiset"
  shutdown_triggered :: bool

type_synonym ('r, 'a) trace = "('r, 'a) state stream"
```

Listing 3.1: State definitions

Each alert `'a` occured on receiver `'r`, which is determined by the `alert_receiver` function. The `is_critical_alert` function distinguishes critical alerts from non-critical alerts. After handling a *critical* alert, a platform shutdown should be triggered. Finally, we fix the maximum number `max_alerts` of alerts that the controller guarantees to handle correctly. These Isabelle definitions are shown in the locale definition of Listing 3.2. Note that our model is independent of the implementations of these functions, we only require them to exist.

```
locale alert_handling =
  fixes
    alert_receiver :: "'a ⇒ 'r" and
    is_critical_alert :: "'a ⇒ bool" and
    max_alerts :: nat
begin
```

Listing 3.2: The `alert_handling` locale

There are five different transitions that can take place on a state:

1. Enabling alerts on receiver `'r`
2. Disabling alerts on receiver `'r`
3. Receiving alerts on receiver `'r`
4. Handling pending alerts
5. (if required) Executing a platform shutdown

Additionally, we allow an idling transition, which keeps the state unchanged. We express these transitions as predicates from one state to another, adding additional parameters when necessary. These are shown in Listing 3.3.

In order to model changes of functions, we use Isabelle's *update* syntax for functions, where `g = f(x := 1)` defines a function `g` that is identical to `f` in every argument except that it maps `x` to 1. A similar syntax exists for records as well, where `t = s(|field := 1|)` denotes a record `t` that is identical to `s`, except for the value of `field`, which is 1.

```
definition enable_alerts' where
  "enable_alerts' s t r ≡
     t = s(| alerts_enabled := (alerts_enabled s)(r := True) |)"

definition disable_alerts' where
  "disable_alerts' s t r ≡
     t = s(| alerts_enabled := (alerts_enabled s)(r := False) |)"

definition receive_alerts' where
  "receive_alerts' s t r as ≡
       size (alerts s + as) ≤ max_alerts
     ∧ t = (if alerts_enabled s r
               then s(| alerts := alerts s + as |)
               else s)"

definition handle_alerts' where
  "handle_alerts' s t ≡
     t = s(| alerts := {#},
             shutdown_triggered := shutdown_triggered s
                 ∨ (∃a ∈# alerts s. is_critical_alert a) |)"

definition shutdown' where
  "shutdown' s t f ≡
       shutdown_triggered s
     ∧ t = s(| alerts_enabled := f,
               shutdown_triggered := False |)"
```

Listing 3.3: Transition predicates between states `s` and `t`

The transitions `enable_alerts` and `disable_alerts` are straightforward. The transition `receive_alerts` is valid iff the number of alerts is at most `max_alerts` and alert receiving is enabled on r, otherwise the state must be unchanged. Handling alerts must consume all of them and trigger a shutdown

```
definition next' :: "('r, 'a) state ⇒ ('r, 'a) state ⇒ bool" where
  "next' s t ≡ s = t
             ∨ (∃r. enable_alerts' s t r)
             ∨ (∃r. disable_alerts' s t r)
             ∨ (∃r as. receive_alerts' s t r as)
             ∨ handle_alerts' s t
             ∨ (∃f. shutdown' s t f)"
```

Listing 3.4: Definition of the next-state relation between two states `s` and `t`

if a critical alert was among the ones processed. Finally, `shutdown` resets the `shutdown_triggered` flag and additionally can en- and disable alerts arbitrarily. The rationale for the latter is that after a platform shutdown, various alerts may trigger (for example a low voltage alert), which are entirely expected, since the platform is shut down.

Combining these state transitions allows us to define a next-state relation, shown in Listing 3.4. It relates two states `s` and `t` iff there is exists a suitable transition between `s` and `t`.

The alert controller's behaviour is then an infinite stream of states, related by the `next'` relation. The corresponding Isabelle/HOL definitions are shown in Listing 3.5.

```
definition "next" :: "('r, 'a) trace ⇒ bool" where
  "next s ≡ shd s = shd (stl s)
          ∨ enable_alerts s
          ∨ disable_alerts s
          ∨ receive_alerts s
          ∨ handle_alerts s
          ∨ shutdown s"

lemma next_iff_next':
  "next s ⟷ next' (shd s) (shd (stl s))"
  unfolding next_def next'_def by blast

abbreviation alert_handler_behaviour :: "('r, 'a) trace ⇒ bool" where
  "alert_handler_behaviour s ≡ alw next s"
```

Listing 3.5: Definition of the alert controller behaviour

Additionally, we define a predicate to characterise the initial states of the system. In an initial state, no alerts have been received and no shutdown was triggered. Which receivers are enabled is arbitrary, and may vary from implemenation to implementation. The `init` predicate is shown in Listing 3.6.

Note that the alert controller behaviour is a predicate on traces, meaning there are potentially infinitely many valid alert controller behaviours. We rely on two properties to guarantee safe behaviour, which we formalised as LTL properties.

```
definition
  init :: "('r, 'a) state ⇒ bool"
where
  "init s ≡ alerts s = {#}
          ∧ ¬ shutdown_triggered s"
```

Listing 3.6: Definition of the initial state predicate

Theorem 1. *Once an alert was received, it stays until it is handled. In other words, no alert is lost. We can express this by the LTL property*

$$\Box\,(\texttt{alerts\_preserved}\;\mathcal{U}\;\texttt{handle\_alerts})$$

where `alerts_preserved` *denotes the property that* `alerts s ⊆# alerts t`.

Theorem 2. *If a critical alert was received, then a shutdown is eventually triggered. We can express this by the LTL property*

$$\Box\,((\exists a \in \texttt{alerts}.\ (\texttt{is\_critical\_alert}\ a)) \longrightarrow \Diamond\,\texttt{shutdown})$$

Proving Theorem 1 is straightforward: it suffices to show that for two consecutive states `s` and `t` either we have that `alerts s ⊆# alerts t` or the transition from `s` to `t` was the alert handling transition. We can express and prove this in a few lines of Isabelle, as seen in Listing 3.7.

```
lemma no_alerts_lost_step:
  assumes
    "next' s t"
  shows
    "alerts_preserved' s t ∨ handle_alerts' s t"
  using assms
  by (cases rule:next'_cases, auto)

theorem no_alerts_lost:
  "alert_handler_behaviour s
     ⟹ (alerts_preserved until handle_alerts) s"
  by (coinduction arbitrary:s rule:UNTIL.coinduct,
      metis)
```

Listing 3.7: No alerts are lost

However, proving Theorem 2 is not possible without liveness guarantees, since the controller can choose to idle eternally. Verification of this theorem relies on a *weak fairness* assumption for the `handle_alerts` and `handle_shutdown` transitions, as is defined in Listing 3.8. Informally, *weak fairness* for a transition `f` implies that if `f` is continually enabled, then it must also occur eventually.

Using these additional assumptions it is possible to prove that once a critical alert was received, the controller will eventually shut down the platform, as proved in Listing 3.9.

```
abbreviation alert_handler_liveness :: "('r, 'a) trace ⇒ bool" where
  "alert_handler_liveness s ≡
       alw (ev handle_alerts) s
     ∧ alw (holds shutdown_triggered impl ev shutdown) s"
```

Listing 3.8: Definition of the alert controller liveness assumption

```
theorem critical_alert_implies_shutdown:
  defines
    "P ≡ λs. ∃a ∈# alerts s. is_critical_alert a" and
    "Q ≡ shutdown"
  assumes
    "alert_handler_behaviour s" and
    "alert_handler_liveness s"
  shows
    "(holds P impl ev Q) s"
proof (rule impI)
  assume 1: "holds P s"
  have "alw (ev handle_alerts) s"
    using alert_handler_spec_def assms by blast
  let ?n = "wait handle_alerts s"
  have 2: "ev handle_alerts s"
    using assms by blast
  define r where "r ≡ sdrop ?n s"
  have "alerts (shd s) ⊆# alerts (shd (sdrop ?n s))"
    using alerts_msubset assms by blast
  then have "alerts (shd s) ⊆# alerts (shd r)"
    unfolding r_def using assms by blast
  then have 3: "holds P r" unfolding P_def
    using "1" P_def set_mset_mono by auto
  define r' where "r' ≡ stl r"
  have "ev Q r'"
  proof -
    have "alert_handler_liveness r'"
      using alw_sdrop assms r'_def r_def by auto
    moreover have "holds shutdown_triggered r'"
      using sdrop_wait handle_alerts'_def
      unfolding r'_def r_def Q_def
      by (smt)
    ultimately show ?thesis using Q_def by blast
  qed
  then show "ev Q s"
    using alw_sdrop not_ev_iff r'_def r_def Q_def
    by blast
qed
```

Listing 3.9: Proof of Theorem 2 under additional liveness assumptions

3.2.1 Refinement

In order for our model to be useful, we have to be able to transfer the security properties we have proven to concrete implementations. We achieve this using *refinement mappings* [19]. A concrete system C with initial state predicate `cinit` and transition relation `cnext` refines our abstract model A with respect to a mediator function $\pi : C \rightarrow A$ if:

1. If `cinit s` holds, then `init (`π`s)` holds
2. If `cnext s t` holds, then `next (`π`s) (`π`t)` holds

Condition (2) is often stronger than necessary [9], and we can weaken it by restricting the concrete states according to some invariant `I`:

2. If `cnext s t` and `I s` hold, then `next (`π`s) (`π`t)` and `I t` hold

An Isabelle/HOL translation of these conditions in shown in Listing 3.10.

```
definition refines ::
  "('c ⇒ bool) ⇒
   ('c ⇒ ('r, 'a) state) ⇒
   ('c ⇒ bool) ⇒
   ('c ⇒ 'c ⇒ bool) ⇒
   bool" where
  "refines I π cinit cnext ≡
       (∀s. cinit s ⟶ init (π s))
     ∧ (∀s. cinit s ⟶ I s)
     ∧ (∀s t. I s ⟶ cnext s t ⟶ next' (π s) (π t))
     ∧ (∀s t. I s ⟶ cnext s t ⟶ I t)"
```

Listing 3.10: Refinement definition

In general, refinement is not complete for proving model simulation, however in our case it is sufficient. Additionally, we prove its soundness in Isabelle/HOL: if a concrete model refines an abstract model, then the guarantees of Theorems 1 and 2 apply to the concrete model as well. The proof is shown in Listing 3.11.

```
theorem refines_sound:
  assumes
    "refines I π cinit cnext" and
    "holds cinit s" and
    "alw (lift_next cnext) s" and
    "alert_handler_liveness (smap π s)"
  shows
    "alert_handler_spec (smap π s)"
proof (unfold alert_handler_spec_def, intro conjI)
  show "holds init (smap π s)"
    using assms by (simp add: refines_def)
  show "alw next (smap π s)"
    by (metis)
  show "alw (ev handle_alerts) (smap π s)"
    using assms by simp
  show "alw (holds shutdown_triggered impl ev shutdown) (smap π s)"
    using assms by simp
qed
```

Listing 3.11: Proof that the refinement relation of Listing 3.10 is sound

Chapter 4

Verification of the seL4 BMC's alert handling implementation

Jetzt geht es ans Eingemachte.

— German figure of speech

In this chapter we introduce the concrete alert handling code of the BMC and how it was verified using Isabelle/HOL and AutoCorres.

Figure 4.1 displays our verification strategy. On top of the stack is the abstract model we defined in the previous chapter, with each successive layer increasing in concreteness until we reach the C implementation. Consecutive layers are connected by some form of correspondence: generation from a lower layer, as detailed on the arrows. White layers denote parts written by hand, while gray layers are automatically generated. Solid arrows denote that a formally verified correspondence proof is produced; this is not the case for dashed ones.

From our C code the C-to-Simpl parser [4] generates an equivalent Simpl

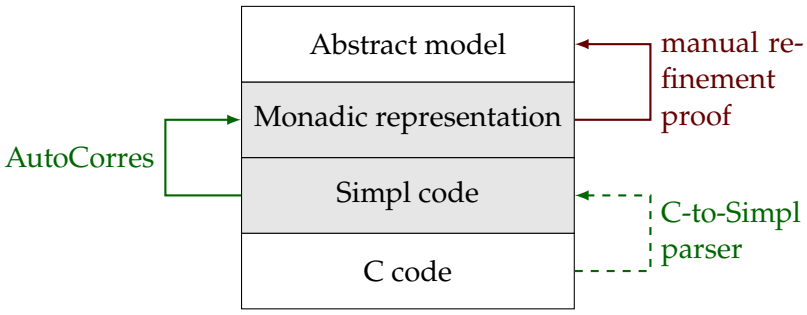

Figure 4.1: Our verification strategy

translation. Note that the C-to-Simpl parser can only parse a subset of C, which is related to the type-safe C0 language [25]. We call the language parsed by the Simpl parser "StrictC" [4]. StrictC's semantics are identical to the C0 language. Restrictions imposed by StrictC include:

- No `goto` statements
- No assignments within expressions
- No fall-through cases
- No unions
- No support for `_Bool`[1]
- and other smaller issues, some of which we encounter below

These restrictions are not as severe as they seem at first glance, as it is easy to rewrite our C code to StrictC code. Problems arise when including pre-written or generated code though, as manual translation is infeasible in these cases. In Section 4.2 we explain how we deal with this problem, as it arises using the CAmkES generated glue code.

Simpl is equipped with a verification framework based on Hoare Logic [25]. However, as we see in Section 4.2, the translated code is tedious to verify, based on the idiosyncrasies of the Simpl language and the structure of the underlying C code.

Thus the AutoCorres [6] tool was developed: it abstracts the translated Simpl code into a monadic representation based on state monads, as explained in Section 2.3. This abstraction occurs in several passes, and for each pass the AutoCorres tool also automatically generates a formally verified soundness proof, showing that the abstraction is sound. Similar to the Simpl language, AutoCorres includes a Hoare logic for its monadic representation and a weakest-precondition tool to aid the verification process.

Ultimately, we want to relate the behaviour of the C functions to transitions taken by our abstract model from Chapter 3, using the refinement relation we specified. The steps we take for a function `f` are shown in Figure 4.2. We separate the reasoning about the behaviour of `f`'s translation from the abstract model. This way, we are free to modify `f`'s implementation or the abstract model with as few adjustments as possible.

This way we separate the proofs about the concrete behaviour of the translated functions and their correspondence with the abstract model, which is more robust against small changes of both the abstract model and the alert controller implementation.

[1] C's official boolean type defines `_Bool` as a keyword for booleans in the C99 standard. It is used in the `stdbool.h` include, which may lead to problems when including other code.

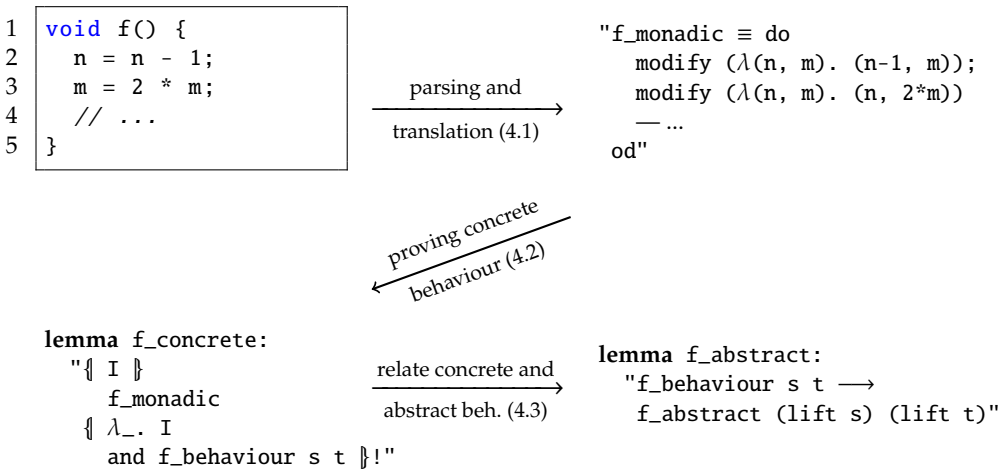

Figure 4.2: Verification steps for proving the behaviour of some function `f`

4.1 Implementing an alert controller for the Enzian platform

The Enzian platform is a dual-CPU research computer for systems research, boasting a powerful, fully programmable FPGA as general-purpose accelerator. The platform features six I^2C buses, on which various integrated circuits are attached that provide a variety of functionality for the Enzian platform: voltage regulators, voltage and current sensors, fan controllers, and more. Each of the six buses has a purpose with respect to the connected devices:

1. The sequencer (SEQ) bus featuring two ispPAC-POWR1220AT8 (ISPPACs in the following), which are general-purpose voltage monitors and power sequencers. The ISPPACs control the enabled signals to the devices on the PWRFAN bus.
2. The power supply (PSU) bus, which is connected to a PMBus/SMBus controller located in the power supply unit.
3. The clock (CLK) bus, which houses three SiLabs clock generators for generating various clock signals on the board.
4. The front panel I/O (FPIO) bus. It is currently unused.
5. The power-fan (PWRFAN) bus, which houses CPU and FPGA voltage regulators and fan controllers.
6. An internal bus of the Enzian BMC SoC, which is irrelevant for our purposes.

On these six buses there are five alert lines, spread across three buses: the SEQ and PSU lines have one alert line each while the PWRFAN bus houses three alert lines. One line attaches to the voltage regulators, denoted PWR, and two lines attach to the fan controller: A regular, unidirectional line `FAN` for alerts and a bidirectional fan fault line `FAN_FAULT`, which can also be used to put the fans in a fail-safe mode.

The Enzian BMC is responsible for driving all ICs on the six I^2C buses of the Enzan platform, as well as correctly handling all incoming alerts. Usually, a SMBus controller has access to the data and clock lines as well as the `SMBALERT` line, if available. However the Enzian BMC, a Xilinx Zynq7000 SoC, does not come with an SMBus controller out-of-the-box. Instead we use the Zynq7000's FPGA to run I^2C and GPIO IPs on the FPGA fabric, which are memory-mapped into the BMC's address space. The I^2C cores drive the data and clock lines for each bus, while the GPIO component monitors the alert lines and signals the controller whenever any are pulled low.

Figure 4.3 shows a rough overview of the BMC and its connections to the main Enzian board. The green component denotes the main BMC component, which is responsible for alert handling on the BMC. The red components denote dependencies of the alert handler.

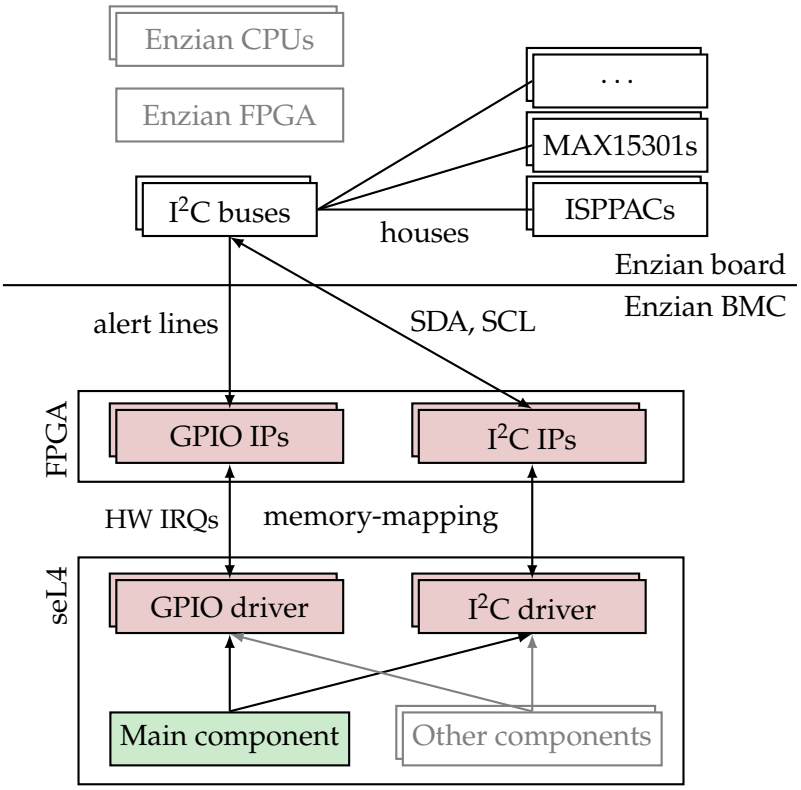

Figure 4.3: Architectural overview of the seL4/CAmkES-based Enzian BMC

We use the same model as Heimhofer [8], whose work we build upon. They provide infrastructure code, such as low-level drivers for the I²C and GPIO IPs, as well as an SMBus driver utilising the I²C driver. We improve on Heimhofer's work by simplifying the low-level driver interfaces and implementing PMBus support for the devices on the Enzian board.

The initial alert handling implementation [8] is part of the main controller component shown in Figure 4.3. Its structure significantly differs from the alert handling model we defined, and thus we implemented our alert handling approach from scratch in StrictC, closely following the model from Chapter 3. For each state transition in the abstract model, we define a StrictC function dealing with the concrete SMBus alert handling. We aim to show that each concrete function refines the corresponding abstract transition.

To store the received alerts we fix a maximum number of concurrent alerts that we handle, and an array of that size. Since we may have potentially received fewer alerts than the array is capable of storing, we also store the number of received alerts. Additionally, we define an enum for distinguishing the alert lines on the BMC, and for each alert line `l`, we track whether alerts on `l` are enabled. This is shown in Listing 4.1.

```
1  typedef struct alert {
2      alert_line_t alert_type;
3      uint8_t device_address;
4      bool has_pmbus_details;
5      pmbus_status_t status;
6  } alert_t;
7
8  #define MAX_ALERTS 100u
9  const size_t max_alerts = MAX_ALERTS;
10 alert_t alerts[MAX_ALERTS];
11 size_t num_alerts = 0;
12
13 typedef enum alert_line {
14   ALERT_LINE_SEQ,
15   ALERT_LINE_PSU,
16   ALERT_LINE_PWR,
17   ALERT_LINE_FAN,
18   ALERT_LINE_FAN_FAULT,
19   N_ALERT_LINES,
20 } alert_line_t;
21
22 bool alerts_enabled[N_ALERT_LINES] = { false };
```

Listing 4.1: Types and global variables for the alert controller

For each alert line `l` on the BMC, we implement the functions `l_receive_alerts`, `l_enable_alerts`, and `l_disable_alerts`. The `l_receive_alerts` function is called whenever the GPIO driver signals that the alert line for `l` is pulled low. A simplified variant of the alert receiving code is shown in Listing 4.2. We omit logging and error checking in this case to better illustrate the core concepts of our alert controller.

Multiple alert lines can be signaled at once or in rapid succession, which may lead to data races and other concurrency bugs. Furthermore, our verification tools do not support concurrent executions, which means that we have to serialise the functions anyway. Thus we introduce a global lock, which each function has to take before it executes and must release after its execution. Hence we can treat each function as running serialised and in isolation.

Next we check the two mandatory conditions for receiving an alert: whether alerts on line `l` are enabled, and whether we have at least one open slot to receive an alert. If any of these checks fail we are allowed to drop the alert silently.

Then comes the core alert handling loop. We know that the `l_receive_alert` is only called when the alert line is held low, so we avoid overhead by not re-checking the alert line unnecessarily. We query the alerting device's address and additional details (if applicable, e.g. for PMBus devices). Finally, we record the alert in our global array and query the current status of the alert

line: if it is still held low and we have more space for alerts, we continue with the next iteration, as shown in Listing 4.2.

```
1  void l_receive_alert(void) {
2      lock_controller();
3
4      if (!alerts_enabled[ALERT_LINE_L]
5            || num_alerts >= max_alerts) {
6          unlock_controller();
7          return;
8      }
9
10     l_bus_lock();
11     uint32_t alert_line_status = 0x0;
12     while (num_alerts < max_alerts
13             && !(alert_line_status & L_ALERT_MASK)) {
14         uint8_t addr;
15         l_get_alerting_device(&addr);
16
17         if (l_has_pmbus_information(addr)) {
18           alert = l_get_alert_details(addr);
19         } else {
20           alert = {.device_address = addr,
21                    .has_pmbus_details = false};
22         }
23
24         alerts[num_alerts] = alert;
25         num_alerts++;
26         alert_line_status = read_alert_lines();
27     }
28     l_bus_unlock();
29
30     notify_alert_handler();
31     acknowledge_alert(L_ALERT_MASK);
32     unlock_controller();
33 }
```

Listing 4.2: SMBus alert receiving loop

After completing the loop we finally notify the alert handler that there are alerts for it to consume and acknowledge that we have handled the alerts for line `l`. This informs the GPIO core that it should signal us the next time the alert line for `l` is pulled low.

Each alert line has a slightly different inner loop, depending on the devices attached to that bus, however they all follow the general structure of Listing 4.2. Note that this implementation requires strict SMBus compliance from all devices on the bus, and may record spurious alerts if some devices do not adhere to the SMBus specification. We encounter and detail two such cases in Section 4.2.1, as well as how we dealt with them.

```
1  void handle_alerts(void) {
2      lock_controller();
3
4      for (size_t i = 0; i < num_alerts; i++) {
5          if (is_critical_alert(alerts[i])) {
6              trigger_shutdown = true;
7          }
8      }
9      num_alerts = 0;
10
11     if (trigger_shutdown) {
12         notify_shudown_handler();
13     }
14
15     unlock_controller();
16 }
```

Listing 4.3: Simplified implementation of the `handle_alerts` function

The `l_enable_alerts` and `l_disable_alerts` functions set the corresponding boolean in the `alerts_enabled` struct. Enabling alerts for a line also acknowledges all pending interrupts on that line, so that the alert handler does not observe stale alerts.

The final two alert handler functions are `handle_alerts` and `shutdown`. Handling alerts is comparatively simple: iterate over all alerts and if a critical alert is encountered, send a signal to trigger the `shutdown` function. The `shutdown` function shuts down all voltage regulators and runs the fans at full speed. The `handle_alerts` function in shown in Listing 4.3.

On the surface, verifying the C implementation seems like a simple problem. However, as described in the beginning of this chapter, the Simpl parser only parses StrictC, a subset of the C language. We have ensured that the alert handler code we wrote is valid StrictC, but unfortunately it turns out that the glue code generated and included by CAmkES is not. We investigated whether it was possible to apply manual patches to the generated code which make it parseable, but it turned out that there were too many issues too deep in the stack, so we had to resort to a different option: supplying *specifications* of CAmkES functions which exhibit the same behaviour as the generated CAmkES code should.

An example is shown in Listing 4.4. The `read_alert_lines` function reads the status of all alert lines as `uint32_t` and returns it. Unfortunately, it is provided by a different component than the alert handler, which means that a CAmkES-generated wrapper function is called instead. As explained at the beginning of this chapter, the generated code is not parseable by the StrictC parser, and thus we have to resort to abstractly specifying the possible behaviours of the `read_alert_lines` function. In principle it is possible to observe any

valid `uint32_t` this way, so we (ab)use a specific feature of the C99 (and also StrictC) language: uninitialised local variables are so-called *indeterminate values*, which means that they can be any inhabitant of their type.

```
1 /*
2  * We intentionally rely on indeterminate values of
3  * uninitialised variables here.
4  */
5 uint32_t read_alert_lines(void) {
6     uint32_t r;
7     return r;
8 }
```

Listing 4.4: Using uninitialised variables to specify nondeterministic behaviour

Inspecting the Simpl translation reveals that `r` is indeed initialised nondeterministically, as shown in Listing 4.5. This is very useful for our behavioural proofs: If `read_alert_lines` instead always returned a fixed value r known to us, then our verification result would be limited to executions where the `read_gpio_values` function returns r.

However, Listing 4.5 also illustrates the shortcomings of the C-to-Simpl translation: the parser tries to be as faithful to the semantics of StrictC as possible during translation, at the cost of additional complexity in the resulting structure.

```
1 TRY
2   lvar_nondet_init r_' r_'_update;;
3   creturn global_exn_var_'_update ret__unsigned_'_update r_';;
4   Guard DontReach {} SKIP
5 CATCH SKIP
6 END
```

Listing 4.5: Simpl translation of `read_alert_lines` from Listing 4.4

The resulting AutoCorres abstraction is much cleaner: it returns a value from the set of all 32-bit integers, as shown in Listing 4.6. In general, the AutoCorres abstractions generated from the Simpl translations greatly simplified the required reasoning.

```
1 read_alert_lines' :: (lifted_globals, 32 word) nondet_monad
2   = select (UNIV :: 32 word set)
```

Listing 4.6: AutoCorres abstraction of Listing 4.5

4.2 Verification of the implementation's behaviour

Finally, we have ensured that all functions defined and called in the alert handling code can be parsed by the Simpl parser. We encountered no issues with AutoCorres' automatic abstraction, and can thus continue with verification of the alert controller's behaviour, as shown in Figure 4.4.

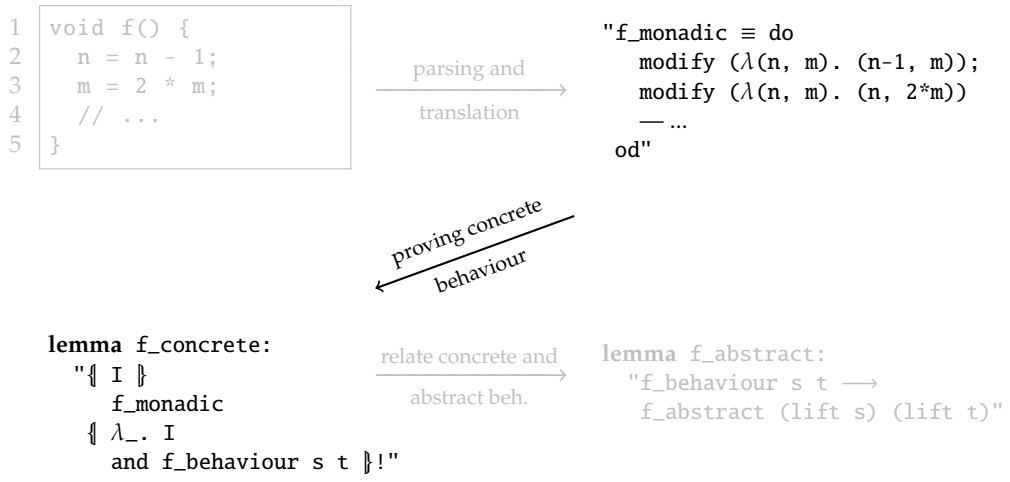

Figure 4.4: Overview of Section 4.2

Note that managing the state of the C code requires explicit synchronisation: the multiset of observed alerts is now described by two variables instead of one, and these two variables must be kept in sync with each other. If, for example, `num_alerts` pointed somewhere outside of the `alerts` array, the concrete program would crash, which would render our safety properties unprovable. Thus we define an invariant `I`, which must always be satisfied in the concrete model.

The two invariants we need are (1) that the number of alerts received is always less than or equal to the size of the alerts array and (2) that all addresses referenced by pointers are always valid. These definitions are shown in Listing 4.7.

For each abstract alert handling function we define predicates on `lifted_globals`, which relate concrete states to each other. As explained before, this simplifies our verification work, as reasoning about the effects of monadic representations on the concrete representation is simpler than additionally lifting the effects to the abstract representation, and more resilient to small changes.

Let us take the representation of the received alerts as a concrete example. The abstract model uses a multiset to describe the received alerts, which has

```
definition
  ptr_inv :: "lifted_globals ⇒ bool"
where
  "ptr_inv ≡ λs. is_valid_w8 s addr_ptr
           ∧ is_valid_w32 s got_ara_response_ptr
           ∧ is_valid_pmbus_status_C s status_ptr
           ∧ is_valid_w16 s status_word_ptr"

definition
  num_alerts_inv :: "lifted_globals ⇒ bool"
where
  "num_alerts_inv ≡ λs. num_alerts_'' s ≤ max_alerts"

definition
  I :: "lifted_globals ⇒ bool"
where
  "I ≡ λs. ptr_inv s ∧ num_alerts_inv s"
```

Listing 4.7: Invariants that must always hold during execution of the C code

no direct equivalent in C. Our representation of the received alerts is the first `num_alerts` elements of the `alerts` array, and instead of showing multiset inclusion of the alerts of some states `s` and `t`, we prove that the alerts arrays of `s` and `t` have identical prefixes. We define the `alerts_prefix_eq` property in Listing 4.8. In Section 4.3, we show how we relate the prefix equality predicate to the subset predicate for multisets in.

This example also illustrates why the invariant `I` is important: without the guarantee that the number of alerts of `t` is at most `max_alerts`, the function `alerts_prefix_eq` could index the `alerts` array out of bounds.

Having formulated the concrete properties and proofs that they imply their abstract counterparts, we set out to prove that the behaviour exhibited by each function matched the property we devised. We rely on the Total Hoare Logic formalized for the state monad by Cock et al. [3] for expressing and proving the desired concrete properties, as well as the included weakest-precondition based VCG.

Verification of the alert controller's AutoCorres-abstracted behaviour was a mostly mechanical task. Based on the weakest-precondition VCG, we formalised and proved successively more complicated total correctness Hoare triples until we were able to tackle all of the defined functions. As we dealt

```
definition alerts_prefix_eq where
  "alerts_prefix_eq s t ≡
      num_alerts_'' t ≥ num_alerts_'' s
    ∧ (∀i < unat (num_alerts_'' s).
        (alerts_'' t).[i] = (alerts_'' s).[i])"
```

Listing 4.8: Definition of alert prefix equality on concrete states `s` and `t`

```
definition abstract_equality where
  "abstract_equality s t ≡
       num_alerts_'' t = num_alerts_'' s
     ∧ alerts_'' t = alerts_'' s
     ∧ alerts_enabled_'' t = alerts_enabled_'' s
     ∧ shutdown_triggered_'' t = shutdown_triggered_'' s"

definition receive_alerts_enabled where
  "receive_alerts_enabled l s t ≡
       alerts_prefix_eq s t
     ∧ alerts_enabled_'' t = alerts_enabled_'' s
     ∧ shutdown_triggered_'' t = shutdown_triggered_'' s"

definition receive_alerts_concrete where
  "receive_alerts_concrete l s t ≡
     if alerts_enabled_'' s.[unat l] ≠ 0
       then receive_alerts_enabled l s t
       else abstract_equality s t"

lemma receive_pwr_alert_behaviour:
  "{| λs. I s ∧ abstract_equality s' s |}
     receive_pwr_alert'
   {| λ_ t. I t
         ∧ receive_alerts_concrete ALERT_LINE_PWR s' t |}!"
```

Listing 4.9: An example of a Hoare triple that we proved

with multiple similar, but not identical definitions, identifying the right trade-off between generalisation and repetition was important in keeping the proof size manageable.

An example of a top-level function for receiving alerts, with its corresponding Hoare triple, is shown in Listing 4.9. The `abstract_equality` function captures the state `s'` at the beginning of the execution of `receive_pwr_alert'`, which allows us to relate the resulting state `t` with the initial state `s` via `s'`. The properties we are interested in are that (1) the controller invariant is preserved and (2) the `receive_alerts_concrete` property holds between the initial state `s` and the resulting state `t`. The proof is quite long and verbose, and can be found in the corresponding Isabelle theory.

This Hoare triple is valid only for the `receive_pwr_alert` function, which receives alerts on the PWR line. We defined similar properties for the other transitions and other alert lines on the Enzian board, until we verified the concrete behaviours of all alert controller functions.

4.2.1 Device quirks and non-compliant devices

Previously in this chapter, we mentioned that the alert handling implementation we implemented requires all devices to strictly conform to the SMBus

specification. Unfortunately, not every device on the Enzian board correctly implements the SMBus alert specification. Listing 4.10 shows the relevant piece of code.

```
1  ...
2  uint32_t alert_line_status = 0x0;
3  while (num_alerts < max_alerts
4          && !(alert_line_status & L_ALERT_MASK)) {
5      uint8_t addr;
6      l_get_alerting_device(&addr);
7
8      ...
9
10     alert_line_status = read_alert_lines();
11 }
12 ...
```

Listing 4.10: Critical part of the SMBus alert handling loop

Correct SMBus alert handling requires that devices correctly respond with their address in line 6, after they have raised an alert. If devices do not respond with their address, then we know that *an* alert has occured, but not *on which device*. One example of a device that violates the SMBus protocol is the MAX15301 [12], a power regulator used to control several on-board voltages: it does not respond to the Alert Response Address (ARA) request after taking the `SMBALERT#` line. Thus it may be possible for us to observe an alert line taken but no device address being returned by the ARA request. We have no choice but to resort to polling all MAX15301s to find out which device is responsible, as shown in Listing 4.11.

```
1  ...
2  while (num_alerts < max_alerts
3          && !(alert_line_status & L_ALERT_MASK)) {
4      uint8_t addr, got_response;
5  -   l_get_alerting_device(&addr);
6  +   l_get_alerting_device(&addr, &got_response);
7  +   if (!got_response) {
8  +       l_poll_max15301s(&addr);
9  +   }
10
11     ...
12 }
13 ...
```

Listing 4.11: Changes when accounting for the non-compliant MAX15301s

The formal correctness proof of the alert receiving function can easily be adjusted to account for these devices. Furthermore, due to the flexible design

of our abstract alert handling semantics, no adjustments have to be made to the abstract model.

Another quirk we observed was some devices seemed to hold the alert line low for a longer time than specified. According to the SMBus specification a device must release the alert line *before* completing its ARA response. However, we observed that the line was held for up to 40ms after the initial request was answered. 40ms is a long time, and lead to about $10-20$ spurious alerts being recorded.

Fortunately, the flexibility of our alert handling model made dealing with this issue simple as well. We can detect a long-held alert line when no device responds to the ARA request and none of the polled devices signals an alert either. The adjustments made are shown in Listing 4.12.

```
...
while (num_alerts < max_alerts
        && !(alert_line_status & L_ALERT_MASK)) {
    uint8_t addr, got_response;
    l_get_alerting_device(&addr, &got_response);
    if (!got_response) {
-       l_poll_max15301s(&addr);
+       l_poll_max15301s(&addr, &got_response);
    }

+   if (!got_response) { break; }

    ...
}
...
```

Listing 4.12: Accounting for long-held alert lines

Not only does this change account for long-held alert lines, but it also gives us stricter termination guarantees. The alert handling code from Listing 4.10 was guaranteed to record an alert in every iteration, potentially recording invalid data, while Listing 4.12 only records valid alert data, and terminates as soon as no more alerts can be read.

Fortunately for us, in this instance we were able to trace the long-held alert lines back to missing pull-up resistors for the alert lines in the FPGA constraints, and this issue vanished after updating the constraint files. However, we kept the adjustments to the alert handling procedure shown in Listing 4.12. It protects the alert controller from devices which exhibit this quirk, and is the overall more defensive option. Adjusting the proofs from Listing 4.11 to Listing 4.12 was non-trivial, since it complicated the loop invariant we had formulated: we need a more sophisticated execution model to handle a potential `break` in the control flow. Nonetheless, we were able to deal with it using local updates instead of rewriting the whole proof.

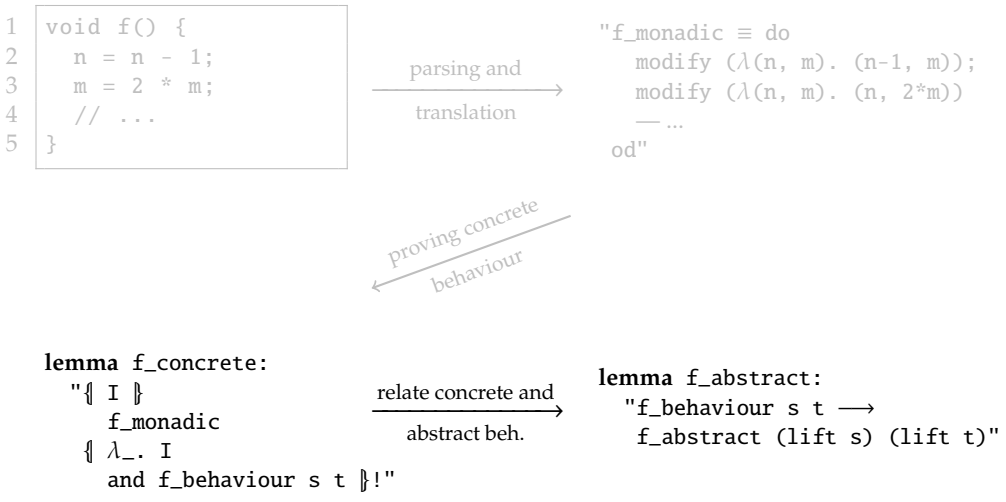

Figure 4.5: Overview of Section 4.3

4.3 Relating the concrete and abstract behaviours

Having proven that the AutoCorres-abstracted code satisfies the concrete Hoare triples we defined, it remains to show that the concrete Hoare triples imply their abstract counterparts, and constructing traces corresponding to the alert handler behaviour, as shown in Figure 4.5.

In order to show semantic equivalence to our abstract alert handling model we need to define a function which lifts the C code's state to the alert handler's state. The AutoCorres-abstracted `lifted_globals` type captures the global variables of the underlying StrictC code, and hence fully describes the concrete state the alert controller is in. The aptly named `lift_c_state`, defined in Listing 4.13 converts the concrete StrictC state to the abstract state representation we chose in Chapter 3.

```
type_synonym alert = "alert_C"
type_synonym alert_line = "32 signed word"

definition
  lift_alerts_enabled :: "lifted_globals ⇒ alert_line ⇒ bool"
where
  "lift_alerts_enabled g s ≡
     if unat s < unat N_ALERT_LINES
       then (alerts_enabled_'' g).[unat s] ≠ 0
       else undefined"

definition
  lift_alerts :: "lifted_globals ⇒ alert multiset"
where
  "lift_alerts g ≡ mset (
     take (unat (num_alerts_'' g))
          (list_array (alerts_'' g)))"

definition
  lift_c_state :: "lifted_globals ⇒ (alert_line, alert) Model.state"
where
  "lift_c_state g = (|
    alerts_enabled = lift_alerts_enabled g,
    alerts = lift_alerts g,
    shutdown_triggered = shutdown_triggered_'' g ≠ 0
  |)"
```

Listing 4.13: Lifting state of the C code to the abstract alert model

Taking the `receives_alerts_concrete` relation from the previous section, we now have to prove that it implies the abstract `receive_alerts` property holds after lifting the controller state. This lemma is shown in Listing 4.14. The proof is quite long and relies on some supporting correspondence lemmas, such as the prefix equality predicate from Section 4.2.

```
lemma receive_alerts_concrete_implies_model:
  assumes
    "I t" and
    "receive_alerts_concrete l s t" and
    "l ∈ alert_lines"
  shows
    "∃as. model.receive_alerts' (lift_c_state s) (lift_c_state t) l as"
```

Listing 4.14: Abstract correspondence proof for the `receive_alerts` behaviour

Let us instead consider the same example as before: proving that the alerts of states `s` are contained in the alerts of state `t`. We have shown that our implementation guarantees that the prefixes of alerts of `s` and `t` are equal. Informally, it is easy to see that if the prefixes are equal, then the (multi-) set of alerts represented by `s` is a subset of the alerts represented by `t`. Proving this formally, on the other hand, requires us to do some work, as shown in Listing 4.15. The whole proof, including its supporting lemmas, is about 80 lines long, which shows that formally proving what our intuition tells us is not always straightforward.

```
lemma alerts_prefix_msubset:
  assumes
    "I t" and
    "num_alerts_'' t ≥ num_alerts_'' s" and
    "alerts_prefix_eq s t"
  shows
    "lift_alerts s ⊆# lift_alerts t"
proof -
  have "take (unat (num_alerts_'' s)) (list_array (alerts_'' t))
          = take (unat (num_alerts_'' s)) (list_array (alerts_'' s))"
    using assms
    unfolding alerts_prefix_eq_def
    by (meson)
  then show ?thesis
    unfolding lift_alerts_def
    using assms(2) take_unat_decomposition by fastforce
qed
```

Listing 4.15: Abstract correspondence proof of prefix equality on `s` and `t`

4.3.1 Executability of the monadic program representation

As mentioned in Section 2.3, proving total correctness of monadic programs does not imply that these monadic programs are also *executable*. Listing 4.16 shows an example of a Hoare triple, which is totally correct but not executable: there is no way to select from an empty set, hence any postcondition is trivially valid. However, there is no state resulting from the execution of `select {}`.

AutoCorres provides a Hoare logic and VCG for executability as well: instead

```
lemma not_executable:
  "{| P |}
     select {}
   {| Q |}!"
  by (simp add: select_def valid_def no_fail_def validNF_def)
```

Listing 4.16: Total correctness of monadic code does not imply executability

of specifying that the pre- and postconditions hold for *all* executions, there must only exist *at least one* execution such that the precondition implies the postcondition. Listing 4.17 shows the definition of `exs_valid`. Note how the ∀ quantifier was replaced by the ∃ quantifier compared to the definition of `valid`.

```
definition exs_valid ("{|_|} _ ∃{|_|}") where
 "exs_valid P f Q ≡
    ∀s. P s ⟶ (∃(r, s') ∈ fst (f s). Q r s')"


definition valid ("{|_|}/ _ /{|_|}") where
  "valid P f Q ≡
     ∀s. P s ⟶ (∀(r,s') ∈ fst (f s). Q r s')"
```

Listing 4.17: Hoare logic for executability of monadic programs

However, the VCG lemmas for `exs_valid` are lacking compared to their total and partial correctness counterparts: many facts we use for the total correctness proofs are not proven for executability. Furthermore, it would require us to duplicate the proof effort we have invested in the total correctness proofs, which is tedious to write and unwieldy to maintain, for example when updating proofs after code modifications.

Fortunately, there is a simpler solution: total correctness of Simpl programs also implies executability, and we can use the AutoCorres-generated correspondence proofs to relate the behaviour of the abstracted, monadic program representation to the behaviour of the equivalent Simpl program, as shown in Listings 4.18 and 4.19. These definitions require us to deal with some implementation details of the Simpl language, such as having to annotate our states as `Normal s` instead of simply `s`. It is still our preferred approach, as it introduces no duplicate proofs.

AutoCorres automatically derives `ac_corres` lemmas for all translated C functions for us. These relate the behaviour of the monadic abstraction `A` to the concrete Simpl translation `C`, according to AutoCorres-derived state translation functions.

We define the relevant definitions of transition on Simpl states and predicate liftings. The lifting process is repetitive and mechanical, but necessary since

we need the executablity property of the Simpl representation to define and prove the refinement relation between the C implementation and our abstract model.

Firstly, we have to construct a concrete variant of the next-state relation introduced in Chapter 3. Listing 4.20 shows the concrete next-state relation we derived for the `receive_pwr_alert` function. These definitions are repetitive boilerplate that is necessary because the AutoCorres' total correctness Hoare triples are not sufficient to prove executability.

Finally, we combine the next-step relations for the individual function into one concrete next-step relation, which we will use as basis for proving refinement of the concrete Simpl and the abstract alert controller model.

We can now prove our main result: that the behaviour of the C implementation refines the abstract alert handling model. This proof consists mainly of a long case distinction, which is why we omit those parts here.

This result allows us to transfer the abstract result of Theorem 1 to the concrete system: it guarantees that our alert controller implementation does not drop alerts as long as less than `max_alerts` have been received.

```
lemma hoaret_from_ac_corres:
  "⟦ ac_corres st ct Γ rx P' A C;
     {| λs. P s |} A {| λrv s. Q rv s |}, {| λrv s. True |}!;
     ⋀s. P (st s) ⟹ P' s; ct
   ⟧ ⟹ Γ ⊢t {s. P (st s)} C {s. Q (rx s) (st s)}"
```

Listing 4.18: Lifting proofs of AutoCorres abstractions to the Simpl code

```
lemma hoaret_executable:
  assumes
    "Γ ⊢t S C T" and
    "s ∈ S"
  shows
    "∃t. exec Γ C (Normal s) t" (is ?P)
    "∀t. exec Γ C (Normal s) t ⟶ (∃t'. t = Normal t')" (is ?Q)
proof -
  have 1: "Γ⊨t/{} S C T,{}"
    by (simp add: assms hoaret_sound')
  from 1 show ?P
    by (simp add: assms terminates_implies_exec validt_def)
  from 1 assms show ?Q
    by blast
qed
```

Listing 4.19: Executablility of total correctness Hoare triples in Simpl

```
definition receive_pwr_alert where
  "receive_pwr_alert s t ≡
     Γ ⊢ ⟨Call receive_pwr_alert_'proc, Normal s⟩
       ⇒ Normal t"
```

Listing 4.20: Next-state relation for the `receive_pwr_alert` function in Simpl

```
definition next' :: "globals myvars ⇒ globals myvars ⇒ bool" where
  "next' s t ≡ s = t
              ∨ seq_alert_enable s t
              ∨ psu_alert_enable s t
              ∨ pwr_alert_enable s t
              ∨ fan_alert_enable s t
              ∨ fan_fault_alert_enable s t
              ∨ seq_alert_disable s t
              ∨ psu_alert_disable s t
              ∨ pwr_alert_disable s t
              ∨ fan_alert_disable s t
              ∨ fan_fault_alert_disable s t
              ∨ receive_seq_alert s t
              ∨ receive_psu_alert s t
              ∨ receive_pwr_alert s t
              ∨ receive_fan_alert s t
              ∨ receive_fan_fault_alert s t
              ∨ handle_alerts s t
              ∨ shutdown s t"
```

Listing 4.21: Concrete next-state relation of the alert controller

```
theorem concrete_refines_abstract:
  "model.refines lift_I (lift_c_state ∘ lift_simpl) init next'"
  (is "model.refines ?φ ?π init next'")
proof (unfold model.refines_def, intro conjI)
  show "∀s. init s ⟶ model.init (?π s)"
  show "∀s. init s ⟶ lift_I s"
  show "∀s t. lift_I s ⟶ next' s t ⟶ model.next' (?π s) (?π t)"
  show "∀s t. lift_I s ⟶ next' s t ⟶ lift_I t"
qed
```

Listing 4.22: Proof that our implementation refines our abstract model

4.3.2 Liveness results

As explained in Chapter 3, proving Theorem 2 (critical alerts eventually lead to a shutdown) is impossible without liveness assumptions on the overall controller behaviour. The seL4 microkernel [14] allows users to run threads using a real-time sensitive scheduler, which we will refer to as the mixed-criticality systems (MCS) kernel. Using the MCS kernel, we could impose real-time restrictions on the runtimes of all alert handling routines and guarantee that our implementation satisfies the required liveness gurantees.

Unfortunately, the Zynq7000 platform, which is the SoC the Enzian BMC runs on, is not supported by the MCS kernel yet. Furthermore, the verification of

the MCS kernel is currently a work-in-progress, and there is no complete result yet. Thus we have no way of formally proving the liveness assumptions of the Enzian alert controller.

Since we cannot prove our assumptions formally (yet), we can instead rely on a informal, but standard argument from scheduling theory, namely rate-monitonic scheduling [21]. A set of n tasks τ_i, each with run-time C_i and unique period T_i is schedulable on a single, real-time processor if the following schedulability test is satisfied:

$$\sum_{i=0}^{n} \frac{C_i}{T_i} \leq n \cdot \left(2^{1/n} - 1\right)$$

In our case, the only functions which require a real-time guarantee are the `handle_alerts` and `shutdown` functions. We estimate the worst-case execution time for both, and thus derive a suitable scheduling period for rate-monotonic scheduling.

The runtime of the `handle_alerts` function shown in Listing 4.3 is clearly dominated by the `for`-loop, and hence the execution time of the `is_critical_alert` function.

Wall-clock timings show that for one alert, the `handle_alerts` implementations takes about 16.3µs. We can easily get a sufficient upper bound by multiplying this by `max_alerts`, which gives us about 1.63ms for 100 alerts.

Timings of the `shutdown` function consistently resulted in less than 3ms, most of which is spent shutting down various SMBus and PMBus devices. Doubling that gives us a conservative worst-case estimate of 6ms.

Scheduling `handle_alerts` with a 6ms period and `shutdown` with a 18ms period thus makes the schedulability test pass:

$$\frac{1.63\text{ms}}{6\text{ms}} + \frac{6\text{ms}}{18\text{ms}} < \frac{2}{3} < 2 \cdot (\sqrt{2} - 1) \approx 0.8284\ldots$$

This implies that rate-monotonically scheduling `handle_alerts` and `shutdown` using the above rates is possible without missing any deadlines, with the remaining time being used to schedule other tasks. Not only does it satisfy the liveness assumptions we made in Chapter 3, but it also suggests a theoretical worst-case alert handling time of 30ms, if we used a rate-monotonic scheduler with periods of 6ms and 18ms.

Until the verification of MCS support in seL4 is completed this is the best we can hope for. Once the Zynq7000 platform is supported by the MCS kernel, we can use seL4's periodic thread scheduling support to assign the above periods to the two functions to at least informally guarantee the necessary liveness properties.

4.4 Reflection

In this chapter we developed and formally verified the Enzian seL4 BMC's alert controller, based on the abstract model presented in Chapter 3. We implement an alert handler in C, which is then parsed and transformed into an representation in Isabelle/HOL using state-of-the-art tools also used by seL4 [14].

We were roughly able to follow the verification strategy depicted in the beginning of the chapter. The only exception is that the monadic program representation, produced by AutoCorres, is not executable. In this case, we had to resort to utilising the lower-level Simpl translation to get the executability results we needed. Figure 4.6 depicts the updated strategy we used.

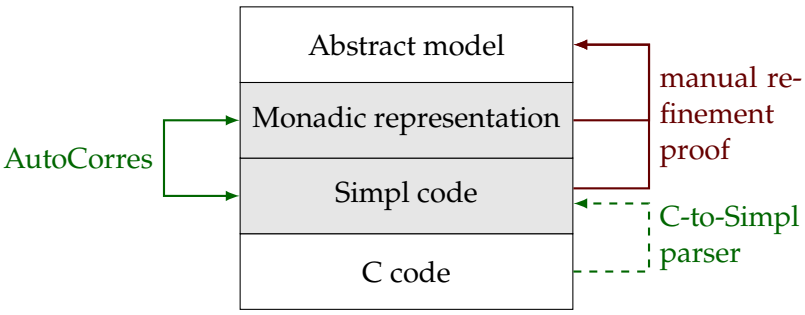

Figure 4.6: Our adjusted verification strategy

Having to resort to combining multiple layers within our verification stack complicates our proof slightly. Breaking the layer between AutoCorres and Simpl adds increases our proof size by about 20%, consisting only of boilerplate definitions and proofs. These boilerplate proofs are fairly simple to maintain, since they are unlikely to change significantly in the future. They do, however, add a cognitive burden to understanding and extending the BMC: it is not possible to understand the proof in its entirety without understanding the basics of the Simpl and its Hoare logics.

One approach to avoid breaking the abstraction layers could be to treat programs that have zero valid executions to have failed. Executability results for deterministic monadic combinators can automatically be derived, and adding a proof obligation to nondeterministic combinators that their result set is nonempty could suffice to prove that our modified total correctness also implies executability.

The Simpl-to-C parser is another component that poses some problems if not used exactly as intended: parsing StrictC code works just fine, however the error messages generated when parsing non-StrictC-compliant C code are cryptic, and determining the root issue requires a measure of familiarity with

the parser to decode. An example is the code shown in Listing 4.23, which uses C's assignments-are-expressions feature, and hence is *not* valid StrictC.

```
1 void f() {
2     int err;
3     if ((err = some_function())) {
4         // do something
5     }
6 }
```

Listing 4.23: C code which cannot be parsed by the Isabelle/C parser

The C parser's error message is "syntax error: replacing YEQ with YSTAR", which is technically correct, but not user-friendly.

Finally, the CAmkES component system itself brings some complications. Most importantly, formal correctness proofs of CAmkES components are currently only possible within a single component, and do not cover the generated glue code which connects the component instances to each other. A verified implementation of the glue code has existed previously, but was scrapped due to performance concerns[2]. There seems to be ongoing work to prove correctness of some connectors, which would help in covering a larger surface area of the CAmkES code. Or perhaps comparing with a seL4-only implementation can shed some light on this question.

[2] https://lists.sel4.systems/hyperkitty/list/devel@sel4.systems/thread/35Y5IJCXTJ4YMADZ4RNLQEHYYYIVRDZY/

Chapter 5

Conclusion

The best validated formal theorem will not guarantee correct behaviour if processor and memory are melting underneath.

— Klein et al., Comprehensive formal verification of an OS microkernel

Starting from a very concrete, SMBus-specific alert handling routine, we developed a general alert handling model for baseboard management controllers. Our model is general enough to cover interrupt- and polling-based alerting, and flexible enough to survive encounter with real-world complications, such as misbehaving devices and hardware issues. Our Isabelle/HOL model is kept small and simple, weighing in at about 300 lines of proof code.

Furthermore, we implemented a concrete alert handler for the Enzian BMC and proved its correctness based on the model we developed. The bulk of our work was spent verifying the alert controller's behaviour, with the resulting proofs covering about 3 500 lines, compared to roughly 700 lines of C code.

We show that a verified implementation of a board management controller is not only possible, but also feasible, using state-of-the-art programming and verification tools. CAmkES' modularisation enables multiple people to concurrently work on verification of disjoint subsystems, which cuts down on the time it takes to verify an entire BMC.

As stated in the epigraph though, we must look beyond what we verified and in the greater context of our work. SMBus devices are supposed to implement a quite rigorous specification, however as we saw in Chapter 4, some devices do not do so correctly. Our implementation is correct with respect to our model of the real world, and can account for minor deviations of

the specification. While we strived to make it as general and simultaneously simple as possible, it is only a model, and must be validated in practice. Unfortunately, formal verification cannot help us here.

5.1 Future work

Our alert handling model is entirely sequential, and depends heavily on isolation for its correctness. Some parts of the alert controller are inherently sequential and cannot run concurrently, for example the communication via I^2C, SMBus and PMBus: due to the electrical implementations of these buses, they can only be accessed by a single device at a time. Other functionality of the alert controller may benefit from concurrency, for example alert retention. Using an array and a past-the-end pointer to index alerts is an effective, albeit primitive way of implementing a producer-consumer scheme. More sophisticated queueing approaches, such as CleanQ [7] for example, exist, and elegantly deal with concurrent modifications, and could be used in the future.

Aside from our model and implementation, there are other, general steps to be taken towards producing formally verified baseboard management controllers.

The alert controller implementation we developed relies on PMBus, SMBus and I^2C drivers to communicate with the peripheral devices. These drivers are unverified, best-effort implementations with no formal semantics or correctness guarantees. There has been some recent work done for I^2C driver generation [11], which could be a way to improve the verification coverage of the BMC.

Alert handling is also not the only aspect that the Enzian BMC controller deals with. It is also responsible for safe power sequencing and device initialisation on boot. As is the case with the I^2C driver, this code is currently unverified and manually tested. Automatic generation of safe power-transitioning sequences, as described for example in [26], would go a long way towards a fully formally verified baseboard management controller.

Using seL4's virtual machines, it would be possible to run an unverified BMC as virtual machine and then isolate and verify the BMC's components step-by-step. Unfortunately, VMs are not supported on the current ARMv7-based Zynq7000 SoC platform used for the Enzian BMC. They would be however on the newer ARMv8 UltraScale+ platform, which would contribute to an improved migration experience from the current OpenBMC-based controller.

Bibliography

[1] Brandon Bohrer et al. "VeriPhy: verified controller executables from verified cyber-physical system models". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 617–630. DOI: 10.1145/3192366.3192406. URL: https://doi.org/10.1145/3192366.3192406.

[2] Christian Svensson et al. *Open-source firmware for your baseboard management controller (BMC)*. Computer Software. 2018. URL: https://github.com/u-root/u-bmc.

[3] David Cock, Gerwin Klein, and Thomas Sewell. "Secure Microkernels, State Monads and Scalable Refinement". In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 167–182. DOI: 10.1007/978-3-540-71067-7_16.

[4] Data61. *C-to-Isabelle parser used in L4.verified*. Computer Software. 2012. URL: https://ts.data61.csiro.au/software/TS/c-parser/index.html.

[5] Nathan Fulton et al. "KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems". In: *CADE*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. LNCS. Springer, 2015, pp. 527–538. DOI: 10.1007/978-3-319-21401-6_36.

[6] David Greenaway. "Automated proof-producing abstraction of C code". PhD thesis. Sydney, Australia: CSE, UNSW, Mar. 2015.

[7] Roni Haecki et al. "CleanQ: a lightweight, uniform, formally specified interface for intra-machine data transfer". In: *CoRR* abs/1911.08773 (2019). arXiv: 1911.08773. URL: http://arxiv.org/abs/1911.08773.

[8] Cedric Heimhofer. "Towards high-assurance Board Management Controller software". MA thesis. ETH Zürich, Feb. 2021. DOI: 10.3929/ethz-b-000490635.

[9] Wim H. Hesselink. "Simulation refinement for concurrency verification". In: *Sci. Comput. Program.* 76.9 (2011), pp. 739–755. DOI: 10.1016/j.scico.2009.09.006.

[10] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[11] Lukas Humbel et al. "A model-checked I^2C specification". In: *Proceedings of SPIN 2021*. July 2021.

[12] Maxim Integrated. *InTune Automatically Compensated Digital PoL Controller with Driver and PMBus Telemetry*. Tech. rep. URL: https://datasheets.maximintegrated.com/en/ds/MAX15301AA02.pdf.

[13] ISO/IEC. *Programming languages — C*. Tech. rep. 1999. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[14] Gerwin Klein et al. "Comprehensive formal verification of an OS microkernel". In: *ACM Trans. Comput. Syst.* 32.1 (2014), 2:1–2:70. DOI: 10.1145/2560537.

[15] Gerwin Klein et al. "Formally verified software in the real world". In: *Commun. ACM* 61.10 (2018), pp. 68–77. DOI: 10.1145/3230627.

[16] Ramana Kumar et al. "CakeML: a verified implementation of ML". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 179–192. DOI: 10.1145/2535838.2535841.

[17] Ihor Kuz et al. "CAmkES: A component model for secure microkernel-based embedded systems". In: *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems* 80.5 (May 2007), pp. 687–699. DOI: 10.1016/j.jss.2006.08.039.

[18] Ph. Lacan et al. "ARIANE 5 - The Software Reliability Verification Process". In: *DASIA 98 - Data Systems in Aerospace*. Ed. by B. Kaldeich-Schürmann. Vol. 422. ESA Special Publication. July 1998, p. 201. URL: https://ui.adsabs.harvard.edu/abs/1998ESASP.422..201L.

[19] Leslie Lamport. "Specifying Concurrent Program Modules". In: *ACM Transactions on Programming Languages and Systems* (Apr. 1983), pp. 190–222. URL: https://www.microsoft.com/en-us/research/publication/specifying-concurrent-program-modules/.

[20] Jochen Liedtke. "On micro-Kernel Construction". In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. ACM, 1995, pp. 237–250. DOI: 10.1145/224056.224075.

[21] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (1973), pp. 46–61. DOI: 10.1145/321738.321743.

[22] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990. ISBN: 978-0-262-63132-7.

[23] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[24] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[25] Norbert Schirmer. "Verification of sequential imperative programs in Isabelle-HOL". PhD thesis. Technical University Munich, Germany, 2006. URL: http://mediatum.ub.tum.de/601799.

[26] Jasmin Schult. "A model-based approach to platform-level power and clock management". BA thesis. ETH Zürich, 2020. DOI: 10.3929/ethz-b-000490632.

[27] NXP Semiconductors. *I^2C-bus specification and user manual*. Tech. rep. 2014. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[28] The OpenBMC Developers. *A Linux Foundation Project open-source Baseboard Management Controllers (BMC) Firmware Stack*. Computer Software. 2015. URL: https://github.com/openbmc/openbmc.

[29] The systemd developers. *The systemd System and Service Manager*. Computer Software. 2010. URL: https://github.com/systemd/systemd.

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):

Towards Trustworthy BMC Software on Modern Hardware

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Fiedler	Ben

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)**
09.08.2021	

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