Proving the Correctness of CANDO3 Optrode Command Interface VDM model in Isabelle/HOL

Leo Freitas*, Ben Wooding*, Bill Scott+, Alastair Pollitt* and Patrick Degenaar=

School of Computing*, School of Medicine+ and School of Electronic Engineering=,
Newcastle University, UK

Abstract. *Background*: implantable medical devices are safety-critical systems. This paper describes the proof of feasibility of a CMOS chip finite state machine (FSM) for an optogenetics brain pacemaker preventing epileptic seizures.
Methods: we use a combination of state-based modelling, theorem proving, model checking, and low-level C program verification. This paper describes the Isabelle/HOL proof of correctness for the FSM only.
Results: we mathematically proved that hidden assumptions were true. A few subtle assumptions turned out to be too strong (*i.e.* impossible proof), which highlighted a rather subtle flaw.
Conclusions: this was a socio-technical experiment: a computing-MSc student with minimal previous experience in formalisms. This convinced stakeholders of formalism usefulness with clearly defined costs in time, effort, and expertise. Results are part of the evidence for the ongoing certification process.

1 Introduction

"Controlling Abnormal Network Dynamics using Optogenetics" (CANDO, `cando.ac.uk`) is a clinically orientated project to develop a new form of brain implant. Its aim is to utilise a combination of gene therapy and optoelectronics to provide closed-loop therapies to aberrant neurological conditions. The first target condition is being developed for focal epilepsy, which affects millions of people worldwide [20].

The approach is radically different . The objective is to continuously control the brain state of the location of brain tissue where the seizures begin (seizure focus) to prevent it from operating outside of a safe domain. CANDO utilises a gene therapy called optogenetics to make brain cells light sensitive using channelrhodopsin [8]

All new techniques create challenges. Neuro-modulator devices have a central control unit , which determines the therapeutic intervention. However, in addition to a central control unit, there is an intelligent brain unit with electronic communication between the two. The brain unit provides optoelectronic stimulation and electrical recording.

This is a multidisciplinary project involving electronic, chemical and material engineering, physics, computing science, medicine and microbiology at multiple academic institutions. The task of formalising the complex electronics became and interesting and difficult challenge given limited documentation. We started participation just before the middle (2016) of the pre-clinical phase (2014–2021). Both rodent and non-human pri-

mate trials are underway. It is hoped that human trials can be expected in the early 2020's during a follow up clinical phase.

This paper focuses on the formal proof in Isabelle/HOL of a VDM model of the CANDOv3 micro chip FSM. With VDM, we discovered three serious problems described in [13]. The work presented here describes the principles of translating VDM to Isabelle and the VDM model proof of correctness. The proof confirmed all findings from VDM, as well as discovered a new subtle issue missed during VDM simulation and coverage analysis, as well as in the C code verification. This was a useful exercise in exploring a combination of different formalisms to ensure the correctness of the low-level C device driver running on embedded hardware that communicates with and commands the cortical implant which applies the optogenetic treatment [13,19].

1.1 CANDO System

The CANDO system (see Figure 1) is composed of two primary intelligent components: i) a control unit placed in the chest; and ii) a brain unit responsible for brain stimulation and recording, as well as performing self-diagnostics. The control unit is responsible for the overall operation, closed loop processing, and communication with the outside world (for monitoring and programming). It is sealed in a hermetic metal canister placed in the chest. The brain unit needs to implement precisely timed stimulus and recording and comprises both local electronics and an optoelectronic array of "optrodes" that penetrate the target brain tissue. The design and implementation of the implantable optrodes are discussed in [3,22]. The area of the brain that the optoelectronic array will be implanted into will have been made sensitive to light via a vector-mediated gene therapy [8].

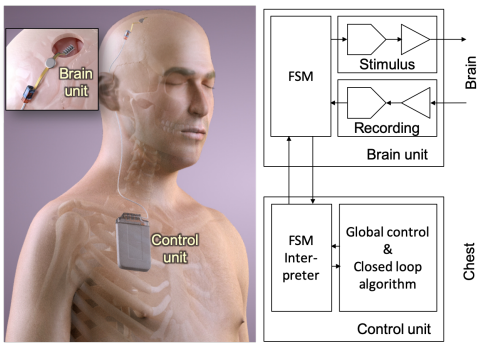

Fig. 1. The CANDO System

The brain unit consists of four optoelectronic arrays of optrodes that are assembled into a single package and implanted into the brain. The brain unit has three primary functions [3,21,22]: i) record neural activity from specific electrodes to amplify and filter that data and convert to a digital data stream to be sent to the control unit; ii) stimulate

from specific light sources with defined intensity and pulse widths as determined by the control unit; and iii) perform diagnostic checks to ensure continued safe function of the brain unit with period data to be sent to the control unit.

Each optrode contains multiple electrodes and LEDs, all of which are controlled by a specially designed complementary metal oxide semiconductor (CMOS) chip with a 24-bit word bus and 17 optrode commands APIs (*e.g.* switching LEDs on/off, switching electrode recording sites on/off, diagnostics, *etc.*). The array, in turn, controls each of its optrodes (*e.g.* synchronicity between multiple electrode recording sites and LED response sites). Each control chip has a FSM, which determines the state of intervention (*e.g.* recording data, transmitting data and optical stimulation and subsets thereof). Global control and closed-loop processing occurs in the control unit situated in the chest (*e.g.* specific focal epilepsy treatment algorithms).

Treatment is delivered through algorithms in the chest unit, which distill to commands for the individual optrode (*e.g.* switch specific LEDs on for specified amounts of time and intensity; monitor/diagnose intended behaviour to ensure expected treatment is delivered, *etc.*), hence delivering the countermeasure to the focal seizure electric spike. For each optrode LED, a crucial safety property is that LEDs cannot stay on for long, as this would cause intolerable temperature differentials [5], and consequent brain function impairments [15]. Other properties exist within optrodes, within the optrodes in the array, and within the array and the chest unit.

As the CANDO project is implementing a medical device with complex control software in a safety critical application, verification is critical to ensure that the optrode command interface is free from errors, operates as expected meeting its specification and ultimately ensuring the safety of patients. However, medical devices certification does not always guarantee safety. In [1], authors report recalls by the Food and Drug Administration (FDA) databases between 2006–2011 and found that 14.7% of recalls were software related. There were 1210 computer-related recalls that affected over 12 million devices. For these computer-related recalls, the FDA found that software failure was the single biggest cause of recall, making up 64.3% of the total. This is despite the fact that medical device software is heavily regulated [2,16].

Related Work

An ambitious and successful attempt at applying formalisms that inspired our earlier efforts was the formal analysis of the Boston Scientific cardiac pacemaker [11]. This was done within the McMaster's mock certification centre initiative (*i.e.* development was done as if under regulatory approval for Canada). To our knowledge, this was the first attempt to tackle the combination of applying formal techniques realistically for industrial-scale certified medical applications. A crucial difference to our efforts, however, is that it was a post-hoc exercise, rather than during actual development. Despite Boston Scientific's involvement throughout the exercise, a challenging (yet common) socio-technical issue emerged: what happens if an error is found? Pacemaker recalls are complicated, and serious perception/financial/legal damage would follow. To solve the conundrum, an earlier version was used in the exercise, where later/current ver-

sions were adjusted to take the use of findings through formal techniques into account. CANDO is similar given its embedded in-vivo nature as a brain implant.

We have successfully applied our approach to other medical devices. In [4], the identification, adaptation, and application of cost-effective industry standard formal techniques with acceptable learning requirements is described for a novel heamodialyser. Many of these results featured in the final CE-marked product (`allmedgroup.com`). Patent protected IP exists, and this machine is now in use within the UK national health service. Finally, in [7], we describe the social-technical issues in using formal modelling and tools within three different certified medical devices.

2 Verification Method

The micro chip architecture (*i.e.* hardware commands, and data layouts) and embedded device driver control software is written in C. The chip fabrication automatically generates 115KLOC of C types, structs, *etc*. The user written (3.1KLOC) code is the actual controller that we modelled and verified. It comprises of three entities:

1. CMOS FSM governing chip's behaviour;
2. CMOS command APIs, assembling instructions packets for each optrode;
3. CMOS main loop, gluing together the FSM and serial communication.

A significant challenge arose from the nature and completeness of the documentation. The CANDOv3 CMOS design was documented via a mixture of circuit diagrams and VHDL code. As an academic project, it was difficult to get better documentation at this stage. As such we had no choice but to start from the C directly.

Understanding the interaction of modules and their dependencies allows the order of verification to be determined, where a bottom-up approach from the C was inevitable. We used VDM as an intermediate language to help hunt for formal specification before verifying the C code [13], which was guided by the workflow in Figure 2. This enabled the capture of various invariants implicitly expected from the FSM and the C code, but that had never been explicitly stated anywhere.

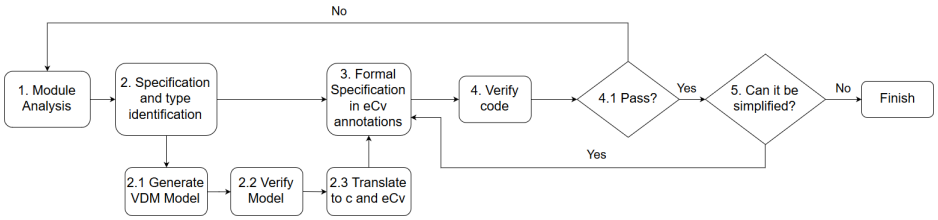

Fig. 2. Verification workflow

Our approach in translating VDM to Isabelle is pragmatic: VDM tools are excellent for modelling, symbolic simulation and test coverage; they have no proof support. Our

interest in proof is related to the feasibility of discovered specifications.We are also interested in some global properties like a chain of operations under certain initial conditions entails desirable outcomes.

One option would have been to write the abstract specification representing the C code directly to Isabelle. We learned from [6] that, even though this is possible given Isabelle's rich libraries, we found ourselves spending considerable effort battling so-called "Isabellisms" (*i.e.* specific Isabelle idioms not always suited to the task at hand). Given the partial (knowledge) and exploratory nature of the modelling exercise, we view the use of VDM (or any other state-based method with sub-typing and partial functions like Z, B or ASM) as adequate. This is useful to explore various modelling scenarios quickly and clearly: modelling decisions are directly related to the abstractions within the original problem, rather than formal languages idiosyncrasies. For example, a type as simple as **`nat1`** is tricky to encode in Isabelle for various technical reasons. As is common among many theorem provers in our experience, one has to adhere to Isabelle's library design decisions, otherwise there is considerable setup and auxiliary proof support effort involved. Arguably, other theorem provers with sub-typing like PVS or CoQ would be a better choice. Nevertheless, we find Isabelle proof automation support invaluable: automatic proof search with **`sledgehammer`**, **`try`**, *etc.*; and counter-example finding with **`quickcheck`**, **`nitpick`**, *etc.*.

Theoretically, we took a narrower logic-view of VDM (*i.e.* left-to-right logic with "possible-semantics" as embedded within the Overture Tools [10]), where LPF plays only a small role. This was based on previous experience in proving VDM theorems within different logics soundly [17,18]. As in this cited experience, our encoding of VDM in HOL entailed theorems proved by Isabelle are valid in VDM. Theoretical details about how that is the case is beyond the scope of this paper. For instance, we use Isabelle's **`undefined`** polymorphic constant to capture various situations where the expected VDM answer is **`undefined`**, yet Isabelle might give something different (*e.g.* the head of an empty sequence in VDM — **`hd []`** — is **`undefined`**, whereas in Isabelle the result is uninterpreted, **`hd [] = hd []`**). In Isabelle, **`undefined`** is effectively a dead end: if it is reached within any proof, no further progress can be made. This suffices to show that there is a problem or missing assumption somewhere in the VDM model translation.

3 Translating VDM to Isabelle

In [6], we present a detailed account on a pragmatic translation strategy from VDM to Isabelle, details of which are beyond the scope of this paper. Since then, we have developed tools to automatically translate from VDM to Isabelle [14].

After logic differences, a key technical aspect of the translation is to account for differences between VDM and Isabelle mathematical toolkit. This is implemented as an Isabelle theory that embeds VDM aspects within Isabelle alongside sufficient automation lemmas enabling proof support. The VDM toolkit has 50 operator and auxiliary definitions alongside over 200 lemmas linking VDM-specific definitions with Isabelle's libraries. This includes numerous bridging interpretations of types from VDM to Isabelle

(*e.g.* VDM sequence indexes start from 1, whereas Isabelle list indexes start from 0) and at same time introducing **undefined** interpretations in Isabelle to VDM expressions where necessary (*e.g.* in VDM `hd[]` returns **undefined**).

There were 13 specific situations when such VDM **undefined** scenarios had to be handled. Our approach here is again pragmatic: we optimise for effectiveness rather than completeness, by addressing operators used in various translation exercises. We define 15 type synonyms and notations as aliases to properly defined types (and their type classes): they do not entail much automation or specification burden. These synonyms serve to provide the Isabelle/VDM user with type names that correspond to VDM types like **nat**, **nat1**, **int**. Other type synonyms include (non-empty) sets, sequences, and maps. For every VDM-type translation, the necessary type invariants are explicitly defined and introduced at the necessary places within the translation. Arguably, a thorough/complete syntactic-driven approach to translation would have been better and we consider this and reporting on design-decisions and proof support as future work. For this paper, we present key principles involved only.

VDM primitive types and constants. Primitive numerical types (*i.e.* **nat**, **nat1**, **int**) need to account for type coercions. VDM performs implicit type coercions, whereas Isabelle expects explicit proof-dependant transformations. For instance, in VDM $(0 - x)$ for a natural number x becomes the value $-x$ of type **int**, whereas in Isabelle $(0 - x)$ equals 0 of type $\mathbb{N}$. We chose to define these VDM types as type synonyms for Isabelle's integer type ($\mathbb{Z}$). This choice for the maximal type mean no type coercions are necessary in Isabelle. On the other hand, type invariants have to be explicitly imposed (*e.g.* every **nat1** variable has to call **inv_nat1**, which is a function from **int** to **bool** stating the value is strictly greater than zero). VDM **real**, **char** and **token** types are translated directly to corresponding Isabelle types $\mathbb{R}$, **char** and **string**, respectively. VDM quote (or enumerated) types are defined with Isabelle **datatype** without type-constructors, whereas VDM union types are defined with Isabelle **datatype** constructors. VDM values are defined as Isabelle **abbreviation**s, which are like **definition**s that do not require manual unfolding during proof.

VDM implicit type checks and coercions. The use of explicitly imposed type invariants is a lightweight (shallow-embedding) approach to the translation: we use the adequate Isabelle type already available. This makes proof support easier to bridge and maintain. It also keeps every implicit check from VDM explicitly given within Isabelle.

Overall, we find this clear and easy to follow and it is not a burden given adequate tool support [14]. Alternatively, we could embed these checks in properly defined Isabelle **typedecl** (*i.e.* axiomatically defined types with non-emptiness proof obligation) or **typedef** (*i.e.* semantic subtypes of existing type) definitions. We had in fact started with this (deep-embedding) approach when first attempting to define VDM's **nat1** as a subtype of Isabelle's $\mathbb{N}$. Given Isabelle's polymorphic type classes, the price to pay is that each new subtype has to be instantiated to the necessary type classes in order to link with underlying definitions and proof machinery. For instance, Isabelle's $\mathbb{N}$ has to instantiate 24 type classes to enable: access to mathematical theories for arithmetic,

linear algebra, *etc.*; and proof support for counter example generation and provably-correct code generation.

VDM set and sequence types. All VDM sets are finite, which is not the case in Isabelle. Thus, a finiteness invariant has to be defined for all VDM sets. Every set element type invariant has to be checked as well (Listing 1). VDM sequences map almost directly to Isabelle lists, where we have to cater for sequence indexes starting from 1 in VDM and 0 in Isabelle and implicit invariant checks for the sequence element type as done for sets. This combination between user defined and implicit checks is pervasive throughout the translation. In fact, to make the translation systematic, we add invariant checks for every type definition (as **`true`**) even when none is required. This made the work of mechanising the strategy much simpler to implement with no onus in proof effort or translation efficiency [14].

VDM maps. Maps are partial and with implicit invariant checks as in sets and sequences. Isabelle defines a map type that is total to an **`option`** type: values outside the map domain relate to **`None`**. In VDM, this effectively means a total map to an optional type (VDM **`nil`**) for values outside the domain. Furthermore, given all VDM sets are finite, all VDM map domains are also finite and must be restricted within Isabelle. Even though this works well most of the time and Isabelle maps provide a number of the necessary VDM toolkit map operators, one still has to deal with Isabelle **`option`** types and its complications associated with function application. An attempt at properly embedding partial maps in Isabelle with type classes instantiations exists: the **`typedef fmap`** defines a finite Isabelle map with all the necessary machinery for its use. We considered using this at first, yet the necessary embedding of VDM type invariants for domain and range elements would entail numerous type class instantiations again. Thus, we decided to keep it simple and use shallow-embedding with Isabelle maps alongside various implicit checks made explicit.

VDM set, sequences and map comprehension and enumeration. Set and sequence comprehension translate directly to Isabelle. Set and sequence expression, binder, generator and filter are available in both languages (*e.g.* {`expr | bind in gen & filter`}), albeit with a slightly different concrete syntax. Like in VDM, Isabelle sequence comprehension generators must be sequences. Set and sequence enumeration are available in Isabelle.

VDM map comprehension is not directly available in Isabelle. Map construction is possible in Isabelle through two lists for domain and range elements that are recursively put together. Practically, we can encode map comprehension through Isabelle's map update over lists operator. These are usually complex and entail difficulties in proof and are better avoided. For instance, one can refactor VDM map comprehension definitions to use VDM toolkit operators instead. Map enumeration is available in Isabelle.

VDM mathematical toolkit. Mathematical operators have to be available within Isabelle, where some are already defined like map domain filtering (<:) or map override (++). We have to define implicit checks for certain primitive types like **`nat1`** being

greater than zero or **set1** being finite and non-empty. We define VDM set cardinality, where infinite sets lead to **undefined**. We also define expected pre and post conditions for various toolkit operators implicitly checked in VDM through the operator's type signature. For instance, VDM cardinality returns a **nat**, which entails the postcondition that its result is always greater than or equal to zero, when viewed as the wider type **int** defined in Isabelle to avoid type coercions.s

More complex VDM operators, such as **let-be-st** or **let-in-set**, can be translated to Isabelle's Hilbert's choice (**SOME**) operator. It can also be used to encode VDM's **iota** operator for definite description. Care needs to be taken in these cases because Hilbert's choice operator is quite difficult to work with during proof.

VDM records. These are translated directly to Isabelle **record**s, where user-defined and implicit record invariants are checked explicitly as done for sets, sequences and maps. An important consideration is that VDM records field access is by projection (`R.x`), whereas in Isabelle it is through functions `(x R)`. That means field names have to be renamed in Isabelle to avoid name clashes. A further complication is that Isabelle records are extensible. This might complicate proof, but it can easily be avoided with narrowing type casts where necessary. Record enumeration is available.

VDM functions. These are translated to Isabelle **definition**s or **fun** in case they are recursive. VDM **measure**s can be useful to inform recursion behaviour, yet do not need to be translated to Isabelle, given its automatic (or user-provided) proof of termination for all recursively defined functions. Furthermore, pre/post conditions are also translated. For the example function `f` below, precondition translation entails the implicit check that: the invariant of type `T` holds for the input `x`; and the precondition of auxiliary function `g` holds. It also entails the explicit encoding of the user-defined precondition `(x > V)` and the call to `g` itself. The same is the case for the postcondition. This is done recursively: implicit type invariants checks for `g`'s input are part of the precondition, as well as any further auxiliary function it might call.

```
1 f: T -> T
2 f(x) = x + 1
3 pre x > V and g(x)  post RESULT > x and h(x);
```

VDM state and operations. State is translated as if it was a VDM record, whereas operations are translated like functions with an extra state parameter and result values. The limitation that only functions (or pure operations) can appear in operation pre/-postconditions is not checked in Isabelle, given only well-formed VDM models are translated. Operation framing conditions (*i.e.* what part of the state remains constant or can change; VDM **ext** read/write clauses) are translated as implicit post condition checks. State initialisation is defined as a VDM constant of state type with explicitly chosen values under the state invariant.

VDM traces and proof obligations. Traces are useful for symbolic execution and specification coverage. They are translated as Isabelle theorems. Similarly, proof obligations

(PO) are translated as boolean-value definitions, which are then stated as theorems. This two-stage definition is important for maintenance reasons: the stated theorem is always the same (*i.e.* the PO name, whereas the definition varies as the specification evolves). Crucially, our primary PO of interest is feasibility: preconditions imply postconditions under universally assumed inputs and outputs given by explicitly defined function/operation calls. That is, in VDM models constructed for executability, all definitions are given explicitly, hence the existential quantification from the feasibility PO can be simplified as:

```
  (forall i : T & pre_f(i) => exists o : T & o = f(i) and post_f(i, o))
= (forall i : T & pre_f(i) => post_f(i, f(i)))
```

4 Technical Considerations

A deep embedding of all VDM types and mathematical toolkit operators with proof support would be ideal. Pragmatically, it is too much work unrelated to the proof task at hand. This is the main reason why we took a shallow-embedding approach.

VDM specification-style for Isabelle proving. To make the explicit checks necessary for VDM type invariants within Isabelle productive, it is useful to follow certain specification guidelines. For instance, the implicit checks for type `Better` below enforce it is a finite set. It also enforces that every element satisfy the invariant for type `T`. This entail another implicit check that values in `T` are natural numbers together with the user defined check they are strictly smaller than 10. These checks are mostly about type invariant compartmentalisation during specification, which entails easier (or more controlled) proofs. For instance, the example below illustrates it is better to have type invariants as close to their related types as possible: even though it is `Ok` to define a set with a universal invariant over its values, it is `Better` to define an invariant on a separate type `T`, which is implicitly checked for all elements of the set type.

```
Ok = set of nat inv s == forall x in set s & x < 10;

T = nat inv x == x < 10;
--@Witness({1,2,3})
Better = set of T;
```

Listing 1. Example better VDM specification-style for proof support

This style is effective for code verifiers like the one we used (eCv) to verify the CANDO C device driver. That is because it helps the code verifier break the problem down, whilst avoiding handling user-defined quantified formulae; a known proof automation bottleneck. Furthermore, VDM allows declaration after use, whereas Isabelle insists on declaration before use: every VDM definition defined this way has to be rearranged during translation. To avoid the visual confusion and complexities involved, we kept to a declaration-before-use style in VDM.

We strongly adhered to the principle that specification clarity cannot be compromised by tool needs. We strive to keep the specification as clean/clear as possible, regardless of any entailed proof complexities. Of course, if equivalently clear definitions are possible with operators with better proof support, this is preferred. Detailed discussion about this is beyond the scope of this paper [6]. This is a crucial part of modelling: if followed correctly, 70-90% of proofs tend to be discoverable or disproved with Isabelle's proof search and counter-example finding support [19]. This has been true of our experience in translating VDM models to Isabelle over the years across numerous domains (*e.g.* payment protocols, garbage collection and memory management algorithms, embedded medical devices, mathematical games, *etc.*). This is not surprising: formal methods proofs are usually numerous yet mathematically shallow, albeit with complex/large expressions.

VDM type witnesses. Existential quantifiers are difficult to automate. To aid this, we extended VDM with proof annotation comments: they enable the specifier to define concrete witnesses for types and function and operation calls. This documents a valid (*i.e.* statically checked invariant preerving) guess for the existentially quantifier witness required during proof. Further details on VDM witnesses annotations are in [9].

5 Results and Discussion

At first, we translated the CANDO CMOS chip VDM model (1040LOC) to Isabelle (1456LOC) manually. Then we proved the 26 theorems about operations feasibility and state initialisation. This required 72 lemmas (822 LOC).

The majority of the proof effort (1 theorem plus 34 lemmas over 430LOC; 52%) was about state initialisation. Optrode command API operation feasibility were next (25 theorems over 155LOC; 19%). The remainder (237LOC; 29%) effort was on 38 lemmas useful for various expressions. This split is not surprising: VDM model complexity is entirely on the FSM type invariant. The 25 command API proofs were mostly the same with minor variations. The state initialisation witness provided by the VDM model annotation helped improve the state initialisation proof.

The VDM model traces had already uncovered the three serious design issues. Nevertheless, two POs failed in Isabelle: one for the `execute` function, which represented in VDM the optrode API function pointers lookup table in C; and the other was for the totalisation of the state initialisation witness, given the C array is dense (*i.e.* every element of the VDM map domain representing the FSM matrix had to have a value). They uncovered a potential retry-reset loop problem within the error recovery command — a situation known to engineers. It also uncovered unknown nuances between the C global variables controlling the optrode device driver SPI message exchanges and how this had to be properly partitioned between 2–3 send/receive commands due to bus (8 bits) vs. message (24 bits) size differences. The proof exercise was crucial to help understand and ensure what was the correct specification for the various FSM commands encoded in the C device driver. That is particularly important given that eliciting such invariants directly from C is very difficult and error prone (*e.g.* C details were mixed within the key hidden abstractions we were hunting for).

Furthermore, C code analysis in [13] led to a discovery of an incorrect timeout being implemented: each FSM state had an associated timeout function that moved to an error state. Thus, it was possible for erroneous code to never allow the loop to terminate. We used the SPIN model checker to exhaustively verify the existence of non-progress cycles (or infinite loops) within a model. We created a SPIN model of the FSM loop, which had a message moving between states to represent the transitions, including the error states mentioned above [19]. A progress and an end label were included to detect infinite looping, as the loop terminates only at one state. This setup would show correct and erroneous termination. This has showed a non-progress cycle at depth 26: it was possible for the loop to run infinitely unless a timeout broke the loop. The given timeout function did not break from the loop but transitioned to another state within it (the error state). Thus, should an erroneous attempt to run the FSM loop be made, then the loop could never terminate. This would be especially dangerous when the chest unit runs treatment algorithms for epilepsy as the patient's health could be directly affected. The correction was trivial: timeout functions within the loop stop it and move onto code outside the FSM loop.

Socio-technical Challenges. We hope to improve on incorporating formal verification into the design process of a multidisciplinary team on a project with funding and timescale constraints. Pre-clinical projects first need to demonstrate efficacy and that is where the predominant scientific effort needs to be placed. Nevertheless, scalability to long-term safe operation is of significant interest. So in balancing these needs, the key perceived issue centred on the opportunity cost: within a defined budget any time not spent by the development team on the primary objective (demonstrating effective operation), could lead to greater project risk. The ideal situation would be for a verification team exploring the secondary objective (demonstrating safe operation).

A key challenge was that the development team, mostly researchers and PhD students in electronic engineering, point blank refused any extra work/learning given the intense milestone delivery timelines. This first round of development from a semi-experienced team would be code from exemplar functions provided by the Microcontroller company. This makes adding MISRA compliance [12] during the development phase a significant extra challenge. It was through convincing the team leader about the research potential (*i.e.* represent the FSM and the assessed risks mathematically in order to enable multiple kinds of analysis), that work was possible. Device driver programmers became curious, and in conjunction with their patience to explain various unclear/undocumented decisions, they engaged with the process. Once that happened, we could see a direct improvement in how the next stages of the C device driver code got implemented: MISRA-compliance, a key characteristic of safety-critical C programs was achievable and the verification process has since been completed. Eventually, the CMOS engineers also got onboard once they "saw" how mistakes could be prevented.

Even though this capability of modelling and proof in uncovering unintended (and often complex/nuanced) behaviours is not surprising to formalists, it was a pivotal moment in the trust-relationship: it changed engineer's demeanour from "impatient skeptics" (*e.g.* why should I bother with the extra work?) to "healthy academic curiosity" (*e.g.* how do you do this? Are there any other issues we missed?). Device driver en-

gineers appreciated the outcomes in terms of helping them identify issues, as well as ensure that potential (error-prone) device driver encoding mistakes were caught as early as possible. Regulatory approval and trial administrators appreciated the emphasis on safety through mathematically precise documentation that enforced design decisions as they evolve and before critical primate and human trials started.

Introducing the process of formal verification to the development team and working with them during the application of the techniques to help develop their understanding was crucial. This transfer of knowledge was accompanied by regularly attending engineering meetings for discussion and presenting work progress, as well as our understanding of the medical device software regulatory process.

To illustrate the possibility of applying this work in practice with a team not solely dependant on a theorem proving expert, we set out the VDM translation and proof as an MSc project [19]. This was risky, yet bounded by the fact we had already manually translated and proved all POs within Isabelle. The VDM translation and Isabelle proof was completed over 5 months, where the involved MSc student had some experience in VDM and Isabelle proof and no prior knowledge of CANDO. This demonstrated that the barrier to entry for an engineer to perform proof requires previous training, but was not too onerous.

The VDM model was translated using an automated tool resulting from a undergraduate student project [14], where 135 POs were generated from a strategy developed by the MSc student. Of these, 130 or 96% of proofs were discovered automatically with Isabelle's **`sledgehammer`**. The remainder 5 proofs had to do with state initialisation and the optrode function pointer lookup table. The latter (failed proofs) highlighted an (obvious) subtlety missed through the manual translation: each function pointer call required specific current optrode state in order to function properly. We missed this over the manual translation because we only proved initialisation from a single initial state (*i.e.* we never allowed the lookup table to be called directly but through the `start` command API). There were 34 of such possible "initialisation options", given the FSM's 34 states; many of which were easily proved given they immediately satisfied the FSM state invariant, apart from these missing 5 cases. Expert time proving these remaining POs was minimal (2 days).

6 Conclusions

This paper summarises the issues in translating VDM to Isabelle and its specific application to the VDM model [13] of a CMOS chip finite state machine (FSM) controlling a novel closed-loop optogenetics brain neuromodulator for epilepsy. We prove feasibility for all CMOS (17) optrode command APIs. State initialisation proof also establishes that the system invariant holds. More details can be found in [19]. These proofs helped ensure the safety properties of a safety-critical medical device that is about to start primate and human trials. The work is now part of the CE-marking certification process. Alongside other work on CANDO's CMOS FSM C device driver [13], it demonstrates how formalisms can be effectively applied for realistic novel medical devices

in practice. VDM extensions, libraries and the Isabelle translator can be found at the VDM_Toolkit github repository[1].

Acknowledgements. We are thankful to Alastair Pollitt, Dimitrios Firfilionis and Ahmed Soltan from CANDO (Wellcome: 096975/Z/11/Z, EPSRC: 102037/Z/13/Z) for their patience and interest. To STRATA (EPSRC EP/N023641/1) programme grant for financial support. Finally, thanks to Nina for inspiration to work with medical devices dependability.

References

1. H. Alemzadeh, R. K. Iyer, Z. Kalbarczyk, and J. Raman. Analysis of safety-critical computer failures in medical devices. *IEEE Security & Privacy*, 11(4):14–26, 2013.
2. BSI. Medical device software — software life-cycle processes. Technical Report BS EN 62304, British Standards (BSi), May 2011.
3. F. Dehkhoda, A. Soltan, R. Ramezani, H. Zhao, Y. Liu, T. Constandinou, and P. Degenaar. Smart optrode for neural stimulation and sensing. In *SENSORS*, pages 1–4. IEEE, 2015.
4. M. D.Harrison, L. Freitas, M. D. snd Jose C. Campos, P. Masci, C. di Maria, and M. Whitaker. Formal Techniques in the Safety Analysis of Software Components of a new Dialysis Machine. *Science of Computer Porgramming*, 175:17–34, Feb 2019.
5. N. Dong et al. Opto-electro-thermal optimisation of optoelectronic probes for optogenetic neural stimulation. *Journal of Biophotonics*, 11(10), March 2018.
6. L. Freitas, C. B. Jones, A. Velykis, and I. Whiteside. How to say why. Technical Report CS-TR-1398, Newcastle University, www.ai4fm.org/tr, November 2013.
7. L. Freitas, B. Scott, and P. Degenaar. Medicine-by-wire: formal techniques for dependable medical systems automation. *Science of Computer Porgramming*, under-corrections, 2020.
8. P. Hegemann and G. Nagel. From channelrhodopsins to optogenetics. *EMBO Molecular Medicine*, 5(2):173–176, Feb 2013.
9. E. Jacobs. Implementing witness annotations for vdmj. Master's thesis, School of Computing, May 2018.
10. P. G. Larsen et al. Vdm-10 language manual. Technical Report TR-001, Aarhus University, Feb 2018.
11. Macedo H.D. et al. Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using VDM. In *International Symposium on Formal Methods*, volume 5014 of *LNCS*, pages 181–197. Springer, 2008.
12. MISRA Ltd. MISRA-C:2012 Guidelines for the use of the C language in critical systems. Technical Report MISRAC:2012, MISRA Ltd, March 2013.
13. A. Pollitt. Verifying the CANDO Project Optrode Command Interface in eCv. Master's thesis, School of Computing Science, Newcastle University UK, August 2018.
14. J. Simm. Creating a Tool for Translating VDM to Isabelle/HOL. Master's thesis, School of Computing Science, Newcastle University UK, July 2019.
15. A. Soltan et al. A head mounted device stimulator for optogenetic retinal prosthesis. *Journal of Neural Engineering*, 15(6), August 2018.
16. U.S. Department of Health and Human Services. General principles of software validation; final guidance for industry and fda staff. Technical Report UCM085281, FDA, Jan 2002.
17. J. Woodcock and L. Freitas. Linking vdm and z. In 13 *ICECCS*, pages 143–152. IEEE, April 2008.

[1] `https://github.com/leouk/VDM_Toolkit`

18. J. Woodcock, M. Saaltink, and L. Freitas. Unifying theories of undefinedness. In *NATO Series D: Information and Communication Security (Marktoberdorf)*, volume 22, pages 311–330. IOS Press, Aug 2009.
19. B. Wooding. Using Formal Methods and Proof to Verify a CANDO Epilepsy Medical Device. Master's thesis, School of Computing Science, Newcastle University UK, June 2019.
20. World-Health-Organization. Epilepsy, Feb 2018. Accessed 08-4-2018.
21. H. Zhao, F. Dehkhoda, R. Ramezani, D. Sokolov, P. Degenaar, Y. Liu, and T. Constandinou. A cmos-based neural implantable optrode for optogenetic stimulation and electrical recording. In *Biomedical Circuits and Systems Conference (BioCAS), 2015 IEEE*, pages 1–4. IEEE, 2015.
22. H. Zhao, A. Soltan, P. Maaskant, N. Dong, X. Sun, and P. Degenaar. A scalable optoelectronic neural probe architecture with self-diagnostic capability. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2018.