This is a post-peer-review, pre-copyedit version of an article published in Science of Computer Programming. The final authenticated version is available online at: https://doi.org/10.1016/j.scico.2019.02.003

Formal Techniques in the Safety Analysis of Software Components of a new Dialysis Machine

Michael D. Harrison[a], Leo Freitas[a], Michael Drinnan[b], José C. Campos[c], Paolo Masci[c], Costanzo di Maria[b], Michael Whitaker[b]

[a]*School of Computing, Newcastle University, Newcastle upon Tyne, NE1 7RU*
[b]*Regional Medical Physics Department, Royal Victoria Infirmary, Newcastle upon Tyne, NE1 4LP*
[c]*Departamento de Informática, Universidade do Minho & HASLab/INESC TEC, Campus de Gualtar, 4710-057 Braga, Portugal*

Abstract

The paper is concerned with the practical use of formal techniques to contribute to the risk analysis of a new neonatal dialysis machine. The described formal analysis focuses on the controller component of the software implementation. The controller drives the dialysis cycle and deals with error management. The logic was analysed using model checking techniques and the source code was analysed formally, checking type correctness conditions, use of pointers and shared memory. The analysis provided evidence of the verification of risk control measures relating to the software component. The productive dialogue between the developers of the device, who had no experience or knowledge of formal methods, and the analyst using the formal analysis tools, provided a basis for the development of rationale for the effectiveness of the evidence.

Keywords: Risk analysis, formal methods, model checking, medical devices, haemodialysis

1. Introduction

The formal analysis of part of a new paediatric dialysis machine, used to contribute to its risk analysis, is illustrated in this paper. The risk analysis was designed to satisfy regulatory requirements (for example [1, 2]). Guidelines typically require an assessment of the hazards associated with a

medical device. These hazards include possible hardware and software failures. Examples of hardware failure include, for example, faulty connections or pump failure. Software failures include both errors due to incorrect design requirements and programming errors.

Conventionally, when a risk analysis is submitted to the regulator it uses test data as evidence that requirements have been satisfied [2]. This is typically an onerous task requiring substantial amounts of test data. Any testing model is likely to be crude because the device is interacting with a physiological system and the cases where the device is used are rare making in-vivo testing very difficult. However it is clear that developing a risk analysis is essential when dealing with life-critical medical systems.

Medical device standards require that measures have been taken to ensure that risks associated with use of the device are as low as reasonably practicable. The required measures include careful identification of hazards and demonstration that risks associated with these hazards have been mitigated. Another part of demonstrating mitigation is to establish requirements of the system that produce barriers between a hazard and its consequence. Processes that are recommended to achieve such confidence include team based scrutiny of the use of documented processes, and software code analysis, such as checking that requirements have been satisfied.

The context of the analysis presented in this paper has important characteristics. It involved a product whose development was close to completion. Its goal was to support risk assessment, adding assurance that the product was safe. Two complementary analysis approaches are described. The first approach involves using formal methods technologies to analyse the design of the software controller of the device. The safety requirements, established in a risk log, were converted into properties that were used in the formal analysis. The analysis provided an adjunct, additional evidence to the testing regime that had been used. As a second approach, techniques were used to analyse the software code, in effect a formal inspection of the code, to identify any vulnerabilities in the way that the software was written. This analysis identified type correctness conditions, and potential vulnerabilities related to shared memory and the use of pointers. These were then assessed using automated proof techniques, in this case through the testing regime or through simulation.

The paper focuses on the role that formal techniques can play in the context of a completed implementation. In this particular case a small team was involved in building and testing the device. If formal techniques are to

be of value within small organisations it is important to understand the stage of the process at which they can be used. An important challenge discussed is whether it is feasible to include a specialist during some or all stages of the development process.

Contribution

We present a pragmatic application of formal methods technologies to the analysis of a new neonatal dialysis machine. The dialysis machine had already been developed pre-product, and the formal analysis provided additional engineering evidence that the product complied with important safety requirements. This included software design level analysis and software source code analysis. The former involved checking that the design specifications were compliant with safety requirements identified by the developers. The latter focused on implementation aspects of the software, such as shared memory, pointers to member-functions and so on. A key aspect of this analysis was also the nature of the interaction between the software developers and the formal analyst.

Organisation

The NIDUS device and its software controller, that forms the basis for the analysis, are described in Section 2. The analysis process is outlined in Section 3. Section 4 describes the tools that were used to assist the formal analysis process. The formal analysis of the NIDUS device is presented in Sections 5 – 9. Related work is in Section 10. Finally discussion and conclusions are presented in Sections 11 and 12.

2. The NIDUS device

Dialysis and ultrafiltration (removal of excess water) are extremely difficult procedures in small children with failing kidneys because the total volume of blood in the child's circulation is very small. The Newcastle Infant Dialysis and Ultrafiltration System (NIDUS) had been used at Newcastle-upon-Tyne's Royal Victoria Infirmary (RVI) experimentally for five years before this analysis was completed [3]. Children had been dialysed previously at the hospital using the same method on an older machine, and before that using a manual technique with syringes based on the same principle. The device does not use a traditional dialysis circuit, and the circuit volume of about 10 mL is suitable for treating infants with a total blood volume less

than 100 mL. Before the device could be used more widely, it was necessary to identify and assess the risks of its use so that the device could achieve regulatory approval.

Dialysing small (< 8 kg) babies is challenging for many reasons. Vascular access for haemodialysis is problematic. The size of the central venous line, required for adequate blood flow, is disproportionately large for the size of the baby. Moreover, current dialysis machines are not accurate enough when managing critical parameters, such as the volume for ultrafiltration and blood volume to be viable for a premature baby. At the time no dialysis machine was approved for use with small children. The lowest limit of viability for the NIDUS device is about 800 grams.

2.1. Architecture of the device

The infant dialyser consists of a number of components including three syringe drives, valves, a bubble detector, and filters. It is depicted in Figure 1. The dialysis process has four stages: *start*, *reset*, *wash*, *dialyse*. The *start* stage differentiates which operational mode the machine will start in either "cold-start" with a new dialysis kit, or "warm-start", where the machine proceeds to the *dialysis* stage bypassing the wash stage. The *reset* stage makes sure that every component of the mechanical apparatus is in its expected positions and are powered up. The *wash* stage runs saline solution through the device to clean the dialysis kit. Finally, the main *dialysis* stage performs a three step control flow that withdraws blood from the baby, filters it according to given parameters, and returns it back to the baby.

The dialyser is formed of various hardware devices controlled by embedded software. This software drives the machine, and communicates with the graphical user interface, which displays current parameters, informs the controller of user key-presses, and animates the physical processes taking place with the involved devices, and with the hardware.

2.2. Software controller: role, design and implementation

The controller software (depicted as a component in Figure 1) detects and warns about error conditions that need attention, as well as issuing hardware interrupts that prevent the machine from behaving in a dangerous manner according to its risk profile. All critical errors (bubbles, clots) are also protected by hardware safety systems. The controller can stop the system, however the core safety of the device is in the hardware. The controller's code is about 4,500 lines of sequential C and C++ code.

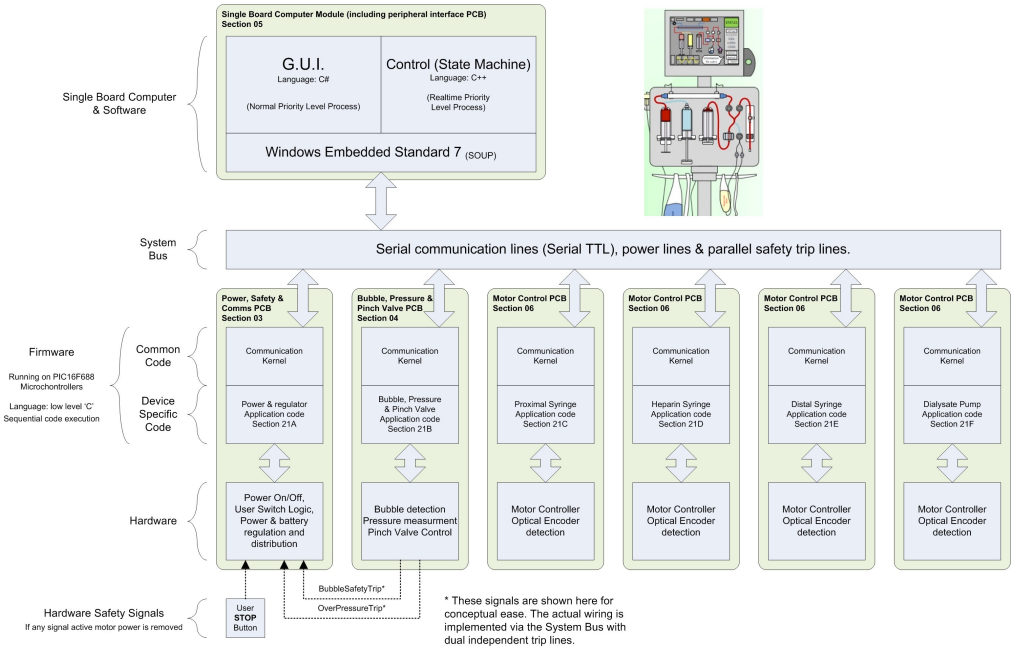
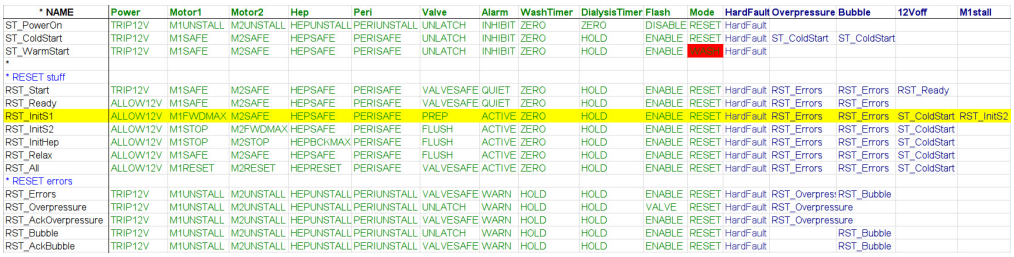


Figure 1: The software architecture

The logic of operation of the controller is organised as a *control table*, and is specified using a spreadsheet (see Figure 2). The control table describes two aspects of the controller. It describes the attributes of the state of the device that control the dialysis process, and it describes how the state of the device changes in response to events. The spreadsheet has been produced by the developers and was used to create the controller's data structure for the software source code. Overall, the table includes 93 states and 30 events.

Each state attribute describes an aspect of the behaviour of the hardware system managed by the controller, for example: *Motor1*, *Motor2* and *Hep* describe the proximal, distal and heparin syringes respectively (values of the attributes include whether the syringe pump is driving forward or backward, and whether "fast" or "slow"); *Valve* and *Bubble* describe the valve assembly and the bubble detectors (the valve may, for example, be safe or open to the baby); *Flash* and *Alarm* describe features of the user interface, for example



* NAME	Power	Motor1	Motor2	Hep	Peri	Valve	Alarm	WashTimer	DialysisTimer	Flash	Mode	HardFault	Overpressure	Bubble	12Voff	M1stall
ST_PowerOn	TRIP12V	M1UNSTALL	M2UNSTALL	HEPUNSTALL	PERIUNSTALL	UNLATCH	INHIBIT	ZERO	ZERO	DISABLE	RESET	HardFault				
ST_ColdStart	TRIP12V	M1SAFE	M2SAFE	HEPSAFE	PERISAFE	UNLATCH	INHIBIT	ZERO	HOLD	ENABLE	RESET	HardFault	ST_ColdStart	ST_ColdStart		
ST_WarmStart	TRIP12V	M1SAFE	M2SAFE	HEPSAFE	PERISAFE	UNLATCH	INHIBIT	ZERO	HOLD	ENABLE	WARM	HardFault				
*																
* RESET stuff																
RST_Start	TRIP12V	M1SAFE	M2SAFE	HEPSAFE	PERISAFE	VALVESAFE	QUIET	ZERO	HOLD	ENABLE	RESET	HardFault	RST_Errors	RST_Errors	RST_Ready	
RST_Ready	ALLOW12V	M1SAFE	M2SAFE	HEPSAFE	PERISAFE	VALVESAFE	QUIET	ZERO	HOLD	ENABLE	RESET	HardFault	RST_Errors	RST_Errors		
RST_InitS1	ALLOW12V	M1FWDMAX	M2SAFE	HEPSAFE	PERISAFE	PREP	ACTIVE	ZERO	HOLD	ENABLE	RESET	HardFault	RST_Errors	RST_Errors	ST_ColdStart	RST_InitS2
RST_InitS2	ALLOW12V	M1STOP	M2FWDMAX	HEPSAFE	PERISAFE	FLUSH	ACTIVE	ZERO	HOLD	ENABLE	RESET	HardFault	RST_Errors	RST_Errors	ST_ColdStart	
RST_InitHep	ALLOW12V	M1STOP	M2STOP	HEPBOLMAX	PERISAFE	FLUSH	ACTIVE	ZERO	HOLD	ENABLE	RESET	HardFault	RST_Errors	RST_Errors	ST_ColdStart	
RST_Relax	ALLOW12V	M1SAFE	M2SAFE	HEPSAFE	PERISAFE	FLUSH	ACTIVE	ZERO	HOLD	ENABLE	RESET	HardFault	RST_Errors	RST_Errors	ST_ColdStart	
RST_All	ALLOW12V	M1RESET	M2RESET	HEPRESET	PERISAFE	VALVESAFE	ACTIVE	ZERO	HOLD	ENABLE	RESET	HardFault	RST_Errors	RST_Errors	ST_ColdStart	
* RESET errors																
RST_Errors	TRIP12V	M1UNSTALL	M2UNSTALL	HEPUNSTALL	PERIUNSTALL	VALVESAFE	WARN	HOLD	HOLD	ENABLE	RESET	HardFault	RST_Overpress	RST_Bubble		
RST_Overpressure	TRIP12V	M1UNSTALL	M2UNSTALL	HEPUNSTALL	PERIUNSTALL	UNLATCH	WARN	HOLD	HOLD	VALVE	RESET	HardFault	RST_Overpressure			
RST_AckOverpressure	TRIP12V	M1UNSTALL	M2UNSTALL	HEPUNSTALL	PERIUNSTALL	VALVESAFE	WARN	HOLD	HOLD	ENABLE	RESET	HardFault	RST_Overpressure			
RST_Bubble	TRIP12V	M1UNSTALL	M2UNSTALL	HEPUNSTALL	PERIUNSTALL	UNLATCH	WARN	HOLD	HOLD	ENABLE	RESET	HardFault		RST_Bubble		
RST_AckBubble	TRIP12V	M1UNSTALL	M2UNSTALL	HEPUNSTALL	PERIUNSTALL	VALVESAFE	WARN	HOLD	HOLD	ENABLE	RESET	HardFault		RST_Bubble		

Figure 2: A fragment of the control table

Flash shows, amongst other displays, that a clip is open or closed, and the Alarm can warn or notify or be quiet. Hence in Figure 2, *RST_InitS1* (identified in the left hand column) is a state that has attributes *Power*, *Motor1*, *Motor2* etc. (top row) with values *ALLOW12V*, *M1FWDMAX*, *M2SAFE* etc. (as described in the row labelled *RST_InitS1*).

The right hand side of the table describes transitions. For example, consider the transition involving *M1Stall* in state *RST_InitS1*. The effect of the transition is highlighted in Figure 2. The event *M1Stall* appears at the top of the last displayed column. The destination state is named in the event column for the *RST_InitS1* row.

In the following sections, the design and source code level analysis of specific aspects of the software controller are presented.

3. The analysis process

Safety aspects of the dialyser were analysed both at the software *design level* and at the software *source code level*. The former involves checking that the design specifications are compliant with safety requirements identified by the developers. The latter focuses on implementation aspects of the software, such as shared memory, pointers to member-functions and so on. The processes are summarised here with reference to the more detailed application to the NIDUS device.

3.1. Design-level analysis process

The design-level analysis includes the following steps (Sections 6 – 8 describe the application of the method to the NIDUS device).

1. The spreadsheet describing the control table of the device was translated into a behavioural model that could be analysed using a model checker.

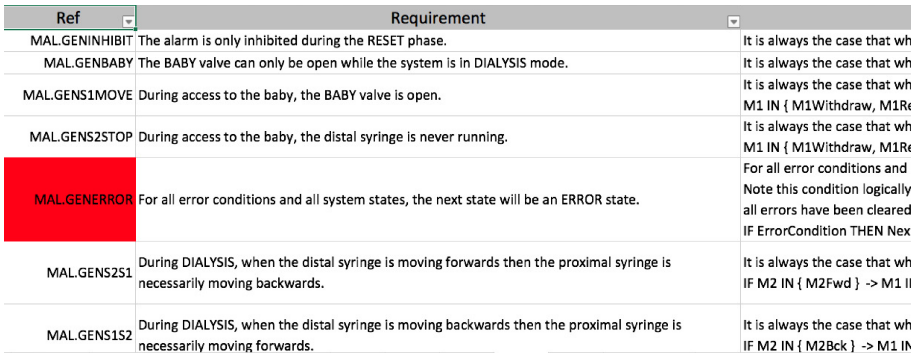
2. Requirements, derived from the risk log, that were relevant to the controller software component, were then considered and expressed in a formal logic.
3. These risk related properties were checked against the model that had been derived from the transition table. Where they failed, discussion with the developers indicated either a flaw in the controller, or a situation which was considered either not to be hazardous, or to be a failure in the formulation of the property.
4. Checking the properties, based on the requirements, often resulted in refinement of the properties or modifications to the control table or the controller mechanism. The results of this process were documented in the risk log.

The behavioural model was specified in Modal Action Logic (MAL) [4], and analysis of the MAL description that was translated into NuSMV [5] and checked with requirements expressed in CTL [6]. The CTL properties were checked of the model using the NuSMV model checker. These facilities are combined together as part of the IVY toolset. A detailed description of the method for generating the MAL model of the control table of the device and for translating the informal requirements into CTL properties are presented in sub-section 4.1.

3.2. Source code analysis process

The process adopted for the design-level analysis includes the following steps (the process is applied to the NIDUS device in Section 9).

1. The main classes used in the code were identified. Functionality for the libraries required by the verifier were developed. While the verifier provides implementations of some of the libraries, it was necessary to develop functionality for other libraries used. This process may lead to minor changes to the code to provide more control over the variables used.
2. The code was checked to be compliant with MISRA guidelines [7].
3. Type and class invariants, as well as pre- and post-conditions for functions and methods were added to ensure that there was no illegal activity; for example, that variables might have unexpected values, or functions computed expected outcomes under the right conditions.
4. Issues detected by the analysis process were checked through simulation to verify that they were not a problem in practice.



Ref	Requirement	
MAL.GENINHIBIT	The alarm is only inhibited during the RESET phase.	It is always the case that wh
MAL.GENBABY	The BABY valve can only be open while the system is in DIALYSIS mode.	It is always the case that wh
MAL.GENS1MOVE	During access to the baby, the BABY valve is open.	It is always the case that wh M1 IN { M1Withdraw, M1Re
MAL.GENS2STOP	During access to the baby, the distal syringe is never running.	It is always the case that wh M1 IN { M1Withdraw, M1Re
MAL.GENERROR	For all error conditions and all system states, the next state will be an ERROR state.	For all error conditions and Note this condition logically all errors have been cleared IF ErrorCondition THEN Nex
MAL.GENS2S1	During DIALYSIS, when the distal syringe is moving forwards then the proximal syringe is necessarily moving backwards.	It is always the case that wh IF M2 IN { M2Fwd } -> M1 II
MAL.GENS1S2	During DIALYSIS, when the distal syringe is moving backwards then the proximal syringe is necessarily moving forwards.	It is always the case that wh IF M2 IN { M2Bck } -> M1 IN

Figure 3: Snippet of the risk log of the NIDUS device

The software code for the dialysis machine was written in a combination of C and C++. The choice of programming language limited the available verification tools that would allow code verification. A proprietary tool, eCv++ (Escher C Verifier [8]), was chosen because it allows the verification of both C and C++ code, and because it provides functional correctness by resolution proofs.

3.3. Dialogue between software developers and the formal analyst

An important element that underpinned all the steps described in the two processes was how the formal analysis was carried out through interaction between developers and analyst. The analysis processes were being performed on the device prior to handing over to a company who would complete the product for example badging and designing the shell and marketing it. The risk log (see Figure 3) had already been constructed and was already being used as a basis for risk analysis.

A testing process had already been carried out and documented to provide evidence to support the assertion that the design satisfied the requirements of the "Standard for Medical Device Software – software life cycle processes" (BS EN 62304:2006 [1]). The risk log provided a source of dialogue between developers and the formal analyst, enabling discussion about the meaning of the requirements and the results of analysis. Risk control measures should be demonstrated to be true, with evidence that is clearly documented, typically, in practice, the results of systematic testing. The BS EN 62304 standard requires a risk analysis path *"from hazardous situation to the software item; from the software item to the software cause; from the software cause to the*

risk control measure; to the verification of the risk control measure". The analysis described in this paper addresses those elements of the risk analysis that relate specifically to the controller component of the dialyser.

4. The verification technologies

4.1. IVY Workbench

IVY is a tool for model-based analysis of interactive systems designs. It is free for academic use, and can be downloaded from `http://ivy.di.uminho.pt`. The tool consists of a set of plug-ins that provide a front end to the NuSMV model checker [5]. The toolkit supports a notation, Modal Action Logic (MAL), that enables the specification of finite state models while at the same time supporting a set of property templates designed to aid the development of appropriate properties for the analysis of the model. The results, which include traces provided by the model checking analysis when a property fails to be true, are visualised. The tool offers a selection of visualisation formats. The IVY tool is designed to provide representations and analysis tools that were more easily usable by developers, and in which the results could be communicated effectively within an interdisciplinary team of software engineers and formal methods experts. The IVY toolkit is organised into an extensible set of interoperable components. These consist of:

- *MAL editor.* This component provides a standard text editor with some support for visualising the structure of MAL models. The modelling language is based on Structured MAL [9]. MAL (Modal Action Logic) is a (deontic) modal logic that incorporates a notion of *action.* MAL describes a logic of actions and is used to write production rules that describe the effect of actions on the state of the device. This style of specification was used because there is some evidence that it is found easy to use by software engineers [10] and was therefore preferred to the notation used by NuSMV. The language also enables the expression of when an action is allowed, using permissions. Non-determinism is possible when more than one action is allowed in the same state of the described model and/or when an action does not fully determine the next state of the system. MAL provides a textual structure similar to the diagrammatic structure of tools such as SCR [11].
- *Property editor.* This component is a front end to the NuSMV model checker [5]. It supports the formulation of properties of the model. Properties are expressed in the CTL notation [6] (which is the notation

used in NuSMV). Assistance is provided in formulating the properties using templates or patterns. The analyst is able to instantiate a property template with the attribute or action names defined in the MAL model.

- *Trace visualizer.* If a property fails to be true of the model a counter-example is produced. This provides a witness indicating one case where the property has failed. The IVY tool enables a variety of formats for these including a matrix notation showing the values of all the attributes of the state at each step of the execution of actions, and a variant of a UML message sequence diagram.

A reason for using this particular toolset was that we were interested in the possibility of producing formal analysis results that would be more easily understandable to an interdisciplinary team. This style of specification had already been found easy to use by software engineers [10]. The goal of IVY is that it should be used eventually without formal methods expertise, and therefore be a key element in communication within the team while at the same time providing the evidence that a requirement under analysis was satisfied.

4.1.1. MAL notation and model generation method

MAL enables the description of attributes to capture the information present in the state of the device and production rules representing actions over these states. The language also enables the expression of deontic operations, in particular permissions were used in our models. Non-determinism is possible when more than one action is allowed in the same state of the described model. Specifically, MAL includes:

- a modal operator $[\_]\_ : [ac]expr$ is the value of *expr* after the occurrence of action *ac* — the modal operator is used to define the effect of actions;
- a special reference event []: $[]expr$ is the value of *expr* in the initial state(s) — the reference event is used to define the initial state(s);
- a deontic operator *per*: $per(ac)$ meaning action *ac* is permitted to happen next — to control when actions might happen.

MAL syntax includes options for defining enumerations, complex state transitions and complex disjunctions and conjunctions of actions. For example:

- $trstate := Power' = ALLOW12V \ \& \ Motor1' = M1STOP$ defines *trstate* as a state transition. Hence it may be used after the occurrence of an action to describe the modification of the attributes *Power* and *Motor1*.

- *modeset* := {*DIALYSE*, *DIALYSING*} defines an enumeration *modeset*.
- *errorevent* := *action1* | *action2* specifies an action that is either *action1* or *action2*.
- *DIALYSE in modeset* is true if *DIALYSE* is a member of the set *modeset*.

Because of these characteristics, MAL constructs allowed a direct translation of the elements of the state transition table of the dialyser. Specifically, the events described in the controller table could be modelled without loss of precision as MAL actions. As an illustration of MAL, the following example declares two boolean attributes that describe whether the device is on (*poweredon*) and whether it is dialysing (*dialysingstate*), and two actions (*start* and *pause*):

interactor *dialyser*
  attributes
    poweredon, *dialysingstate* : *boolean*
  actions
    start pause
  axioms
    [*pause*] !*dialysingstate′* & keep(*poweredon*)
    per(*pause*) → *dialysingstate* & *poweredon*

This simple example model describes the effect of the action *pause* as setting the attribute *dialysingstate* to false and leaving the attribute *poweredon* unchanged. Priming is used to identify the value of the attribute after the action takes place. A permission predicate restricts the *pause* action to only happen when the system is dialysing and powered on. The *keep* operator preserves the value of the attribute *poweredon* in the next state. If an attribute is not modified explicitly or is not in the *keep* list, then its value in the next state is left unconstrained.

4.1.2. CTL notation and requirements translation method

Properties are presented for analysis via the IVY property editor. The notation used was CTL as supported by the NuSMV model checking tool. A full description of CTL can be found in, for example, [6]. CTL provides two kinds of temporal operator, operators over paths and operators over states. Paths represent the possible future behaviours of the system. When p and s are properties A(p) means that p holds for all paths and E(p) that p holds for

at least one path. Operators are also provided over states: G(p) means that p holds for all the states of the examined path; F(p) that p holds for some states over the examined path; X(p) means that p holds for the next state of the examined path while [p Us] means that p holds until s holds in the path. CTL allows a subset of the possible formulae that might arise from the combination of these operators. AG(p) means that p holds for all the states of all the paths; AF(p) means that p holds for some states in all the paths; EF(p) means that p holds for some states in some paths; EG(p) means that for all states in some paths; AX(p) means that p holds in the next state of all paths; EX(p) means that p might hold in the next state; A[p Uq] means that p holds until some other property q holds in all paths; E[p Uq] means there exists a path in which p holds until some property q.

The approach adopted for translating natural language requirements into CTL follows the guidelines described in [12]. Key notions and logic relations are first identified in the natural language requirements. The identified notions are mapped to state attributes of the MAL model. Relations are translated using CTL operators. Property templates [13] are used as guidance to facilitate a correct translation. These steps can be partially automated, e.g., using heuristics [14] thereby guaranteeing correctness. In this analysis however, the translation process was performed manually. Future work will produce automated support for the instantiation process.

4.2. Tools used for the source code analysis

The Escher C++ verifier (eCv++) [15] was used. It combines formal verification (i.e., total functional correctness with resolution proof) with partial MISRA compliance checking. eCv++ is based on C and C++ semantics extended by abstract types and constructs from the Perfect Developer tool [16]. It accommodates C++ features including classes, templates and varied casting operators. Functional correctness can be verified using the software design-by-contract approach, where pre/post conditions as well as type invariants, are given to classes, types, and functions. These contracts are written in an extended C++ language, and the tool ensures, through verification conditions (VCs), that the code for the chosen compiler and target architecture delivers the given contracts. VCs are proved using a combination of term rewriting and a resolution/para-modulation automated theorem prover, with a detailed and structured audit proof trace. This is also useful for the purposes of certification. Other useful eCv++ features include detailed contract suggestions upon verification condition failure (that

is, counter-examples are already described in terms of possible contracts). Some simple (yet common) contracts are also automatically identified by the tool. Verification conditions generated by eCv++ were analysed using the eCv++ theorem prover and the Isabelle/HOL Word theory.

5. Translating the NIDUS state transition table into a formal model

The controller spreadsheet was translated systematically into MAL using the method described in sub-section 4.1.1. After the initial development of a translator, subsequent models were generated automatically from a CSV file (provided by the developers) representing the controller. The translation strategy is described in [17]. The aim was to ensure that the MAL model represents the finite state model, as described by the spreadsheet, accurately. Overall, the translation of the control table of the device into MAL, including meta-attributes, involved 682 lines, including 119 lines of state definitions and 152 lines of type and constant definitions.

The controller software assumes that a pipeline of events exists, each tick of the system process causes the next event to be taken from the pipeline. The default transition appears in the last column of the spreadsheet (not shown). If the pipeline is empty then a specified default transition is taken. The model does not capture the pipeline of events explicitly. It simply models the "next event" as an action, and assuming that the pipeline may become empty, includes an explicit default action. This simplification is sufficient to capture most of the properties of the events but assumes nothing about loss or corruption of information in the pipeline. These conditions must be considered separately. When several actions are possible, because the guard for each of them is satisfied, then one of the actions is taken non-deterministically. In this way, all possible behaviours of the device are considered. There are some circumstances where it is necessary to prove properties that assume that the pipeline is never empty. To analyse these situations an additional attribute was added to the model that becomes false if a default action occurs in a path.

A concrete translation example is now demonstrated. Consider the effect of the transition related to event *M1Stall* highlighted in Figure 2. The event *M1Stall* appears at the top of the last displayed column. The state *M1Stall*, from which the event transitions to *RST_InitS2*, is described in the eleven columns that follow the name *M1Stall*. The destination state is named in the event column for the *RST_InitS1* row. The events described in the

development of the controller could be modelled without loss of precision as MAL actions. The MAL description of the effect of *M1Stall*, when the event occurs in state *RST_InitS1*, is as follows:

$$statedist = sdRSTInitS1 \rightarrow [acM1Stall]\ trRSTInitS2$$

The expression above indicates that, when the state is *RSTInitS1* the action *acM1Stall* leads to the state *RSTInitS2*. The attribute *statedist* has a type that enumerates a set of "labels" indicating each state of the control table. If *RSTInitS1* is the current state then *statedist* takes the value *sdRSTInitS1*. Hence the model defines a set of transformations that change current state to specified new states. In the MAL model, a predicate *trRSTInitS2* is used to abbreviate the set of state attribute transitions associated with changing the state to *RSTInitS2* as described in the following MAL definition.

$$\begin{aligned} trRSTInitS2 := {} & Power' = ALLOW12V\ \&\ Motor1' = M1STOP\ \& \\ & Motor2' = M2FWDMAX\ \&\ Hep' = HEPSAFE\ \& \\ & Peri' = PERISAFE\ \&\ Valve' = FLUSH\& \\ & Alarm' = ACTIVE\ \&\ WashTimer' = ZERO\ \& \\ & DialysisTimer' = HOLD\ \&\ Flash' = ENABLE\ \& \\ & Mode' = RESET\ \&\ statedist' = sdRSTInitS2 \end{aligned}$$

This transformation specifies new values for each of the attributes, for example the value of the attribute *Power* becomes *ALLOW12V* etc.

The model was further extended by adding more state attributes that deal with features of the controller that are only listed as comments in the spreadsheet. They enable the analysis of further features of the user interface that were transformed by the specified events. These features include the attributes associated with the soft function keys for the function keys *key1*, *key2*, *key3*, *stop* as well as the audible alert. Hence *trRSTInitS2* has further characteristics in the model:

$$\begin{aligned} & seclr' = GREEN\ \&\ !audiblealert'\ \&\ fkey1' = F1BLANK\ \& \\ & \quad fkey2' = F2BLANK\ \&\ fstop' = F3STOP \end{aligned}$$

6. Deriving requirements from the NIDUS risk log

This section summarises the negotiation involving the developers and the analyst, using the risk log described in Figure 3. It also considers the range of types of requirement that are contained in the risk log. This discussion is expanded in the next section.

6.1. Refining a sketch requirement as a CTL property

The risk log contains a list of requirements developed to mitigate known hazards. An example of a requirement in the risk log, described in Figure 3, is requirement MAL.GEN2S1:

> "During DIALYSIS, when the distal syringe is moving forwards then the proximal syringe is necessarily moving backwards."

The developer produced a partial translation of this in discussion with the analyst into a pseudo formal expression. This formulation provides an indication of the developer's understanding of the requirement without noting the temporal dimension of the property or the precise nature of the sets $\{M2Fwd\}$ and $\{M1Bck\}$.

> If M2 in {M2Fwd} → M1 in {M1Bck}

The two sets $\{M2Fwd\}$ and $\{M1Bck\}$ were then articulated by the formal analyst as MAL definitions:

M2FWD := { M2FWDMAX, M2FWDUNUF }
M1BCK := { M1BCKMAX, M1BCKUF, M1WITHDRAW }

It was confirmed by the developers that these attribute states comprised all those relating to forward and backward motion in the two motors. Having defined the relevant state attributes as specified in the spreadsheet model, the next step was to formulate a precise version of the property as a basis for the analysis. Some risk log requirements were difficult to express in CTL, and produced complicated CTL expressions that were difficult to explain to the developers. For example, there were several requirements in the risk log that required that given some past event a certain property will not hold. These properties were simplified by adding further attributes to the model that had been automatically generated. The modifications will be discussed in the next section in the specific case of requirement category P4.

6.2. Categories of requirements

Several categories of requirement were identified as the risk log was considered. These are now listed before considering examples in more detail.

P1: States are required to be inaccessible in dangerous circumstances. Examples include the property outlined above in Section 6.1. A further example found in the risk log is that: "it should not be possible to dialyse an infant with heparin in the blood circuit".

P2: Normal behaviour as represented by "normal events" should always lead to the same sequence of events. An example from the risk log is that "when the dialysis machine is error free it always generates a correct dialysis sequence including appropriate wash and dialysis stages".

P3: An error state should always lead to an appropriately safe action. The risk log contained requirements of the form: "when an error event occurs then the device is taken to an appropriate error state".

P4: States can only be reached if combinations of states have happened in the past. An example of such a property is that relevant reminders are always displayed to "close a clamp before the next phase of the cycle can commence".

7. Checking NIDUS' risk related properties

Specific requirements will be considered in detail in this section. A set of 252 requirements was identified in the risk log of which 47 mitigations used the MAL analysis at least in part. The analysis involved 23 properties. These supported mitigations relating both to aspects of protection and design. Verifying all the properties together on a MacBook Pro with Intel Core i5 clocked at 2.9GHz, with 8GB RAM and SSD memory, took 1.7 seconds.

P1: Unsafe combinations of states cannot occur. The requirement discussed in Section 6.1 falls into this category:

> "During DIALYSIS, when the distal syringe is moving forwards then the proximal syringe is necessarily moving backwards."

The requirement was formulated in CTL as:

$$AG(Motor2\ in\ M2FWD \rightarrow Motor1\ in\ M1BCK) \tag{1}$$

The property checks that for all states, when *Motor2* is in a forward state, then *Motor1* is in a backward state.

	1	2	3	4	5	6
main.action		acDefault	acKey2	ac12Voff	ac12Von	acM1stall
Alarm	INHIBIT	INHIBIT	QUIET	QUIET	ACTIVE	ACTIVE
DialysisTimer	ZERO	HOLD	HOLD	HOLD	HOLD	HOLD
Flash	DISABLE	ENABLE	ENABLE	ENABLE	ENABLE	ENABLE
Hep	HEPUNSTAL	HEPSAFE	HEPUNSTAL	HEPSAFE	HEPSAFE	HEPSAFE
Mode	RESET	RESET	RESET	RESET	RESET	RESET
Motor1	M1UNSTALL	M1SAFE	M1UNSTALL	M1SAFE	M1FWDMAX	M1STOP
Motor2	M2UNSTALL	M2SAFE	M2UNSTALL	M2SAFE	M2SAFE	M2FWDMAX
Peri	PERIUNSTAL	PERISAFE	PERISAFE	PERISAFE	PERISAFE	PERISAFE
Power	TRIP12V	TRIP12V	TRIP12V	ALLOW12V	ALLOW12V	ALLOW12V
Valve	UNLATCH	UNLATCH	VALVESAFE	VALVESAFE	PREP	FLUSH
WashTimer	ZERO	ZERO	ZERO	ZERO	ZERO	ZERO

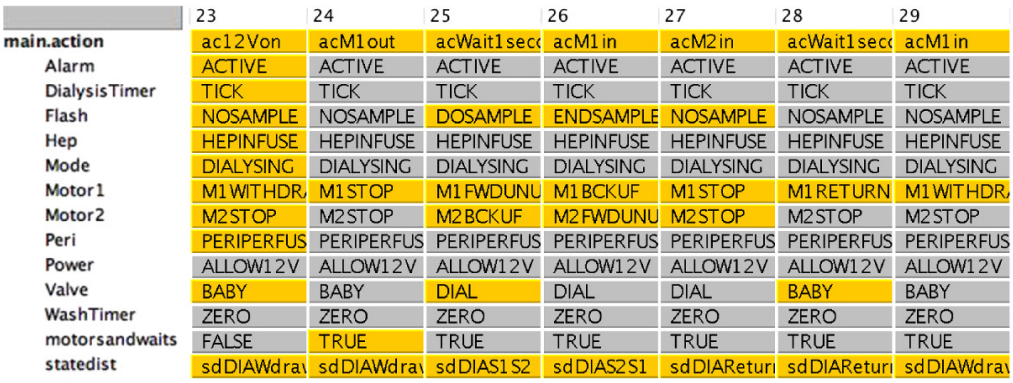
Figure 4: Counter-example to property P1

Attempts to prove the property of the model failed. The counter-example (shown in Figure 4) indicates one path in which it fails. The figure shows a sequence starting from an initial state (column 1), ending at a state where the property fails (column 6). Columns indicate values held by attributes. These are named in the left hand column (i.e., column 0). For example, the attribute *Power* has value *ALLOW12V* in column 4. The colour yellow is used to indicate that a state attribute has changed value between successive states. The path indicates (as shown in the row marked *main.action*) that from the initial state the device defaults (that is it takes the action *acDefault*) because there are no events in the queue. This action is followed by *Key2*, followed by *12voff*, *12von* and *M1stall* which leads to the state where the property fails.

Discussion during the risk meeting explored the implications of the sequence and came to the conclusion that this exception was acceptably safe and could therefore be excluded. The considered property was therefore refined by excluding this case, and the analysis continued. Several other cases were found where the property failed. The risk analysis team considered each of these exceptions and noted that the common property of these counter-examples was that they occurred when the device was not in dialysis mode, hence the following property was constructed:

$$AG((Motor2\ in\ M2FWD\ \&\ Mode\ in\ \{DIALYSE, DIALYSING\}) \rightarrow Motor1\ in\ M1BCK)$$

The property formulated as a result of this observation is true for the model. It should be noted that this observation about the exceptions was a problem of formulation, the property could be expressed more simply. It could be



	23	24	25	26	27	28	29
main.action	ac12Von	acM1out	acWait1sec	acM1in	acM2in	acWait1sec	acM1in
Alarm	ACTIVE	ACTIVE	ACTIVE	ACTIVE	ACTIVE	ACTIVE	ACTIVE
DialysisTimer	TICK	TICK	TICK	TICK	TICK	TICK	TICK
Flash	NOSAMPLE	NOSAMPLE	DOSAMPLE	ENDSAMPLE	NOSAMPLE	NOSAMPLE	NOSAMPLE
Hep	HEPINFUSE	HEPINFUSE	HEPINFUSE	HEPINFUSE	HEPINFUSE	HEPINFUSE	HEPINFUSE
Mode	DIALYSING	DIALYSING	DIALYSING	DIALYSING	DIALYSING	DIALYSING	DIALYSING
Motor1	M1WITHDR	M1STOP	M1FWDUNU	M1BCKUF	M1STOP	M1RETURN	M1WITHDR
Motor2	M2STOP	M2STOP	M2BCKUF	M2FWDUNU	M2STOP	M2STOP	M2STOP
Peri	PERIPERFUS	PERIPERFUS	PERIPERFUS	PERIPERFUS	PERIPERFUS	PERIPERFUS	PERIPERFUS
Power	ALLOW12V	ALLOW12V	ALLOW12V	ALLOW12V	ALLOW12V	ALLOW12V	ALLOW12V
Valve	BABY	BABY	DIAL	DIAL	DIAL	BABY	BABY
WashTimer	ZERO	ZERO	ZERO	ZERO	ZERO	ZERO	ZERO
motorsandwaits	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
statedist	sdDIAWdra	sdDIAWdra	sdDIAS1S2	sdDIAS2S1	sdDIAReturi	sdDIAReturi	sdDIAWdra

Figure 5: Proving the 'dialysis cycle'

argued that visual inspection of the spreadsheet would have been sufficient to indicate the problem in this particular case. However this systematic approach to finding paths to potentially hazardous states provides an exhaustive approach. At the same time it makes it clear to the team the circumstances in which the property fails.

P2: Staying in the dialysis cycle. The second class of requirements relates to ensuring that the behaviour of the device follows the required cycle in normal circumstances. The requirement is expressed as follows:

> "MAL.DIALCYCLE: Unless there are errors or user actions, the system stays in the dialysis cycle"

The requirement aims to ensure that, barring error events or user interventions, the transition table will always cause the device to complete the same haemodialysis process. To check the requirement it is first assumed that no such cycle exists. This property should fail and give, as an example of failure, one cycle that is of the appropriate form. Once the cycle is discovered, and is judged to be the correct 'dialysis cycle', the next stage is to show that it is the only possible cycle that can be generated using the relevant actions.

Team discussion clarified that the state was intended to start at state *DIA_Wdraw* with starting event *M1out*. The normal cycle involves specific events (*M1in*, *M2in* and *Wait1second*) recognised as drivers of the dialysis cycle. An attribute is introduced *motorsandwaits* that is set to true for the first event of the cycle and whose truth is preserved only by events: *M1in*, *M2in* and *Wait1second*. This limits attention to the relevant events, any

other event sets the attribute to false. The following property was used to find a sequence that satisfies these constraints.

$$\begin{aligned} AG(statedist = \ & sdDIAWdraw \rightarrow \\ & AX(AG(!(motorsandwaits \ \& \\ & statedist \ = \ sdDIAWdraw)))) \end{aligned} \tag{2}$$

This property should generate the cycle as a counter-example. This is illustrated in the trace fragment in Figure 5.

The sequence fragment starts with the state *DIAWdraw* (bottom row, column 23) when *motorsandwaits* is false and ends with the state *DIAWdrawRlx* (column 29). The sequence of actions that make up the cycle are shown in a row at the top of the table (*acM1out* etc.). This sequence is indeed the 'dialysis cycle' as acknowledged during a meeting. The value of *motorsandwaits* is shown in the penultimate row and remains true throughout. This counter-example identifies *one* cycle only. It does not exclude the possibility of others and therefore it is necessary to check that the sequence is unique. This is easily done by proving that the only next step, using these events for each state in the sequence, is the one identified in the counter-example. For example, properties of the following type must be true:

$$\begin{aligned} & AG(statedist = sdDIAWdraw \rightarrow \\ & \quad AX(motorsandwaits \rightarrow statedist = sdDIAWdrawRlx)) \end{aligned}$$

This process was completed and the cycle, using the specified events, was found to be unique.

P3: Errors lead to error states. An important issue in the risk log was to ensure that error events would always lead to error states, expressed for example as:

> "MAL-GENERROR: For all error conditions and all system states, the next state will be an error state."

Furthermore it was required that the device should remain in an error state if further error events occur. The error events were defined using the following MAL definition:

$$ErrorEventSet := acHardFault \mid acOverpressure \mid acBubble \mid acPeriStall$$

The elements of *ErrorStateSet* were then identified and the following property agreed to capture the requirement:

$$AG(AX(ErrorEventSet \rightarrow statedist\ in\ ErrorStateSet))$$

The property was found to be false because the Alarm can be inhibited and, when this happens, the property fails. This possibility was therefore excluded:

$$AG(Alarm\ !=\ INHIBIT \rightarrow \\ AX(ErrorEventSet \rightarrow statedist\ in\ ErrorStateSet))$$

This property was also false and therefore further refined. An error state had wrongly been included in *ErrorStateSet* and it was also recognised that in a particular *Mode*, namely *RESET* the property would fail.

$$AG(Alarm\ !=\ INHIBIT\ \&\ Mode\ !=\ RESET \rightarrow \\ AX(ErrorEventSet \rightarrow statedist\ in\ ErrorStateSet)) \tag{3}$$

All these exclusions were judged acceptable. None of them were considered to compromise the safety of the device. The property however continued to be false. The state *STWarmStart* also required exclusion. This exception was not an error state but it was not considered to be problematic because its occurrence is clear and does not cause confusion to the clinical operator. This iterative process led to successive weakening of the original property. Each step involved discussion and clearly documented rationale as to why the exception could be reasonably excluded. The rationale became part of the safety case documentation.

P4: States can only be reached if combinations of states have happened in the past. An important set of requirements deals with reachability properties in the context of some previous combination of states. For example, that specific information should be presented to the user to remind them of the requirement for a specific action. A concrete example is:

> "MAL.HEPCLIP: The user is instructed to close clip before changing syringe, and re-open afterwards."

This particular instruction is represented by the controller using the *Flash* attribute. The *Flash* attribute identifies dialyser warning displays. For example, $Flash = HEPCLOSE$ indicates that "close the heparin clip" has been transmitted. There are a variety of ways in which this property can be expressed using CTL.

We chose to adopt a simple approach that could be explained easily. A meta-attribute *hepclipopen* is true when $Flash = HEPOPEN$ is the last display relevant to the clip. The attribute becomes false when *Flash* has values *HEPCLOSE* or *HEPSYRINGE*. For example, the following fragment involving the *Hepin* action specifies a transition to the state *HEPClip*. This state includes a change to the *Flash* attribute $Flash' = HEPCLOSE$ and therefore *hepclipopen* is set to false. It is assumed therefore that the clinician will recall the last display relating to the opening or closing of the clip.

$$\begin{array}{l} statedist\ in\ \{sdDIAReady\} \rightarrow [acHepin] \\ \quad trHEPClip\ \&\ !motorsandwaits'\ \&\ keep(\ldots)\ \&\ !hepclipopen' \end{array}$$

We then check the property:

$$AG(Mode = DIALYSING \rightarrow hepclipopen)$$

The property proved to be false because a particular state *HEPPrime* provides a clear alternative indication to open the clip that does not involve the flash display. This rationale was included in the risk assessment and the model was changed so that the meta-attribute was also set to true when visiting *HEPPrime*. The property then becomes true.

8. Documenting the risk log

The original risk log, see Figure 6 for a fragment, indicates a range of potential risks across the hardware and software of the device.

The risk log is organised as columns. For example the sixth column indicates the type of control proposed to mitigate the hazard. This is indicated in the first column as: "blood loss". The fifth column indicates "the method of control" which specifies a barrier that mitigates the hazard. Three of the methods of control relate to the controller and indicate requirements on the controller designed to mitigate the hazard (Figure 3 indicates how some of these methods of control were elaborated). MAL.GENBABY refines the

Blood loss	PREP pinch valve erroneously OPEN during dialysis, or operator does not fit tube correctly to PREP valve.	During ultrafiltration or return cycle blood may be returned to waste bag.	M10.5	In all DIALYSIS states, PREP valve is closed.	Controller	
			M10.6	Manual clamp added to prep and waste line.	Mechanical	
			M10.7	Instruction on computer display to close clamp after wash cycle is complete. Users need to acknowledge done before operation can continue.	Controller	
			M10.8	Blood in waste bag should be evident to operator. Guidance about clip in instructions for use.	Info	
	PATIENT pinch valve	A mixture of blood	M10.9	In all WASH states, BABY valve is closed.	Controller	

Figure 6: Extract from the original risk log

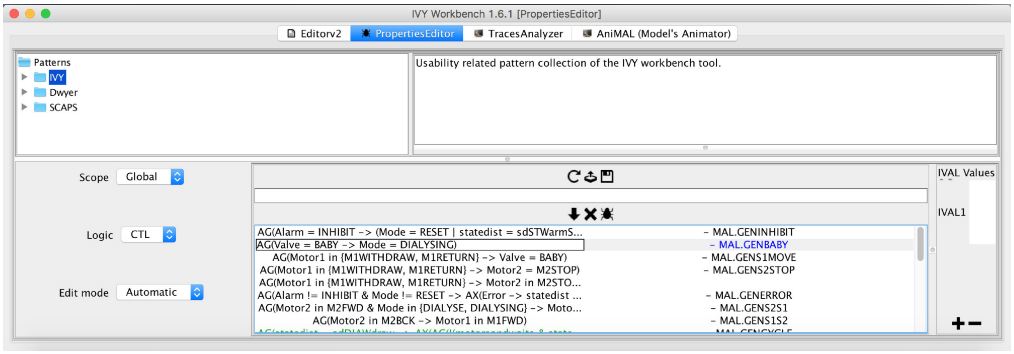


Figure 7: The MAL.GENBABY seen in the IVY property editor

bottom case (M10.9) on Figure 6. The barrier is refined as the controller requirement "the baby valve can only be open while the system is in DIALYSIS mode". This actually strengthens the original requirement because WASH mode is one of several modes that are supported by the controller. The right hand column of the revised risk log (Figure 3) indicates a semi-logical expression of the requirement. The reference to MAL.GENBABY refers to evidence produced by the IVY tool (see Figure 7).

In this way there is a trail from the original method of control, contained in the risk log, to the requirement expressed as a property proved of the model. The MAL model, and list of properties along with the IVY tool are therefore part of evidence that along with test data is designed to provide confidence that the requirement is satisfied.

9. Performing the source code analysis of NIDUS

9.1. Identification of the main classes of the code

Modules were first identified and organised, focusing on the dependencies between them. This process was mainly performed manually, with help from Microsoft's Visual Studio call graph tools. Alternatively, commercial tools like GammaTech CodeSonar[1] can be used for more complex program infrastructures. The modules were organised as an ordered acyclic graph: cycles were identified and dependencies were minimised by reorganising given code across modules. The second step was to identify library (and OS-dependent) boundaries, in other words to find dependencies on C or Windows libraries. Some OS features would require shadowing to enable the analysis process, and some re-definition was necessary, for example to change C++ strings to C null terminated arrays. Shadowing involved developing alternative `.h` files to redefine necessary library types and APIs with code contracts that enable verification of their use. It was important that these assumed specifications did not compromise results. For this reason, implementations were also provided to some of the key library APIs, where assertions and exceptions were used, to ensure that at least, if specifications are violated, the program could be taken to a safe albeit exceptional state. The third step was to find top-level control loops and to mark modules that interact with them. It was necessary to follow these dependencies to the least dependent parts, finding the leaves of the graph. In some cases the call graph tool of Microsoft Visual Studio was used to discover these links. The fourth step was to find the most used and least dependent leaves and identify which ones are most important to the target problem of interest, which were the calculations used to determine the outcome of the control table events, which in turn determined what next state the machine should go to.

9.2. Checking compliance of the code with the MISRA guidelines

Focus was given to key aspects of the code, such as the control state machine, environment sensed data, and device controller modules. The dialyser contains side-effect free functions. These functions are responsible for various calculations using sensed data, for example representing the exchange

[1] https://www.grammatech.com/products/codesonar

of sensed pressure from the blood circuit for motor speeds driving the syringes. These are crucial and ultimately responsible for the conditions guarding events and state changes from the dialyser state machine. Given their central role and relative isolation from other parts of the code, it was the most effective target of functional correctness verification beyond MISRA compliance and memory safety checks. Dependencies relating to standard libraries, such as `stdio` and `std::string`, were identified. Whenever possible, arrays of characters were preferred rather than `std::string` in order to avoid issues with dynamic memory allocation. Efforts were made also to shield against architecture-specific code, such as OS-specific libraries and OS or architecture dependent bit-vector operations.

Changes were made to the code to make it MISRA compliant. MISRA guidelines are divided into three categories: *mandatory*, where deviation is not permitted; *required*, where any deviation must be recorded and authorised according to provided deviation procedures per MISRA rule; and *advisory*, which are recommendations that often cannot be ignored, but ultimately do not constitute a breach of compliance. Checking differences between code subsets helped highlight where MISRA compliance identified potential issues. Moreover, running the MISRA compliance tool within eCv++ for the original code also highlighted various issues of interest. In practice, approaching compliance makes it easier to discharge the eCv++ verification conditions and the simpler the code contracts become.

With a clearer understanding of what MISRA rules would be involved for the key parts of the dialyser's code, we started looking at what (least invasive) changes to the original code would be possible. Often compliance changes were easy to achieve. This process led to a dialogue with the development team, raising awareness of the kinds of issues such compliance would avoid, and creating two positive outcomes: the recognition that the code was close to MISRA (mandatory rules) compliance and some (but not substantial) work was needed for further MISRA (required rules) compliance. Moreover, compliance also entailed simplified VCs as the space of potential breach was considerably reduced as the example below demonstrates.

9.3. Pre- and Post-conditions

All constants involved in bitwise operators were made explicitly unsigned (e.g., `0x07` becomes `0x07u`) as required by MISRA *Rule 7.2*. This is important to ensure congruent results across C compilers and target linked-program

execution-architectures. This entails a considerable simplification in generated VCs because value range checking for multiple operations is no longer necessary. Moreover, all MISRA *Essential Type Model* rules (10.1-10.8) are also required and impose, among other options, no type widening (required Rule 10.6). Through this process the following (original) code:

```
inline int foo(int MSB, int LSB) {
    int Moverlap = MSB & 0x07 ;
    int Loverlap = ( LSB >> 5  ) & 0x07 ;
    int DeltaOverlap = ( Loverlap - Moverlap ) & 0x07 ;
    ...
}
```

Listing 1: The original code

is transformed through MISRA compliance with some of the required rules to the code fragment described in Listing 2.

```
inline unsigned int foo( unsigned int MSB, unsigned int LSB ) {
  unsigned int Moverlap = MSB & 0x07u ;
  unsigned int Loverlap = ( LSB >> 5u ) & 0x07u ;
  unsigned int DeltaOverlap = ( Loverlap - Moverlap ) & 0x07u ;
  ...
}
```

Listing 2: After MISRA compliance modifications

Types are tightened to avoid unsigned values, and mask constants are made explicitly unsigned (i.e., `0x07u` instead of `0x07`). This simplifies C type cast widening and narrowing rules involving bitwise operators (e.g., shift left widens the result to the closest memory size needed for the resulting expressions). Further transformation is required to conform to MISRA's Essential Type model rules. For example, the subtraction result may be negative. Casting is required to keep within the same unsigned type domain. This creates a further problem because such casting would betray other rules about bitwise operations over negative numbers. These kinds of change require more thought and potentially substantial change to the code. The following verified version (Listing 3) includes full functional correctness contracts (pre/post/invariants). Preconditions, together with input type invariants, ensure that inputs are within $0 \ldots 255$ and that their deltas under the corresponding three key bits (i.e., highest three bits for the LSB, and lowest three bits for the MSB) are within a certain boundary (i.e., maximum three bits apart or equal, hence the $-4 \ldots 3$ range).

```
typedef unsigned int invariant(value in 0..255) Byte;
inline unsigned int foo( Byte MSB, Byte LSB ) const
  pre ( ( LSB >> 5u ) & 0x07u ) - ( MSB & 0x07u ) in -4..3
  post result in 0..(12<<2)-1
{
  Byte Moverlap = MSB & 0x07u ;
  Byte Loverlap = ( LSB / 32u ) & 0x07u;
  Byte DeltaOverlap = 0u;
  if (Loverlap >= Moverlap) {
    DeltaOverlap = (Loverlap - Moverlap) & 0x07u;
       ...
  } else {
    DeltaOverlap = (Moverlap - Loverlap) & 0x07u;
    ...
  }
  ...
}
```

Listing 3: Verified version

An example VC, where the unsigned type restriction has been avoided would include checks like `minof(int) <= X`. With the unsigned modification, the number of VCs would be halved. A concrete example is given in Listing 4.

```
(minof(int) <= (63u & msb)) &&
(minof(int) <= (63u & (lsb / 4u))) &&
(((63u & (lsb / 4u)) < (256u + (63u & msb))) &&
 ((63u & msb) <= (63u & (lsb/4u)))) &&
(minof(int) <= (63u & ((63u & (lsb/4u)) - (63u & msb)))) &&
(minof(int) <= ((63u & ((63u & (lsb/4u)) - (63u & msb))) + msb))
&& (((63U & ((63U & (LSB / 4U)) - (63U & MSB))) + (63U & MSB))
       in minof(int)..(maxof(int)-1))
```

Listing 4: Verification condition

This is the eCv++ suggested precondition for the original code (Listing 1), which is complex and difficult to understand. The adherence to MISRA led to an initial simplification of this VC. Isabelle/HOL [18] was used to normalise and simplify it. This effort was enough to bring the problem back to the developers in a way that led to the documentation of the actual contract (Listing 3) that made sense to developers and was sufficient to discharge all the involved VCs.

The issue was raised with the developers as it related to the implemented code. The code transformations performed by the tool provided a reference

point that was used to discuss the issue, and exhaustive testing was used to ensure code modification kept faith with the original. At the same time the modifications prevented the bad outcomes from inputs, possible in the original code, that betrayed the precondition. In this case "exhaustive testing" meant that all 65353 inputs were used in the testing process. A program was constructed that ran the original and the modified code on all inputs and collected their differences as a spreadsheet. The differences showed which input values led to erroneous outputs. In this particular case, an exhaustive analysis was feasible, because of the small size of the state space. In general, for algorithms with infinite or very large input states, code coverage criteria enable the minimisation of the number of meaningful test cases with respect to the underlying specification, at the cost of losing exhaustiveness in the analysis.

9.4. Simulations and tests

As a result of the verification carried out with eCV++, various coding issues were found. In particular, bitwise operators, when compiled to different targets, could create serious problems depending on the interpretation of signed values. Differences could allow for dangerous behaviours, or inaccurate handling of error situations. Key findings included 8 potential bugs, two of which could lead to serious outcomes if compiled to different targets in the future. The risk assessment team judged these situations, supported by test and simulation, to be safe for the present product. They flagged it as an issue for any future implementation. The contracts for these dialysis-related functions had the effect of highlighting the potential issue of signed integer overflow under certain bitwise and arithmetic operators. The conversation within the risk assessment team flagged future improvements in the robustness of the code.

As a result of this analysis process 50 coding issues were revealed. These included, for example, stricter use of type signage, as well as about 20 minor issues, such as improved naming conventions. The annotated and slightly modified code totalled 7,500 lines. The eCv++ tool generated 782 VCs, of which 721 were discharged automatically and relatively quickly (e.g., in a couple of minutes). 38 of the remaining VCs, could not be discharged by the eCv++ theorem prover, although information was provided in terms of a proof trace. These VCs were mostly related to bitwise expression patterns and floating point numbers. Some of these led to improvements in eCv++ prover itself. These improvements reduced the list of "unprovable" VCs. The

remaining unproved VCs used Isabelle/HOL Word theory when the designers felt they were crucial.

The original code contained C/C++ idioms that eCv did not handle. Discussion with the tool implementors helped elicit either alternatives (e.g. when the original had coding patterns compromising mandatory MISRA checks) or in most cases extensions. Extensions were both syntactic (e.g. for C: #define for bit constants, enum types with explicit values, etc; C++: object member function pointers, links with Windows.h APIs, etc.), and semantic (e.g. extended rules about floating point arithmetic precision for the operations involved; improved hints of expected preconditions, etc.).

Given the nature of the problem, VCs were mostly similar in nature (e.g. various bit masks over unsigned char), if different in actual structure. When underlying expressions were complex, the failed VCs would contain about a quarter of a page of the whole expanded/rewritten expression. The Word library was used to handle bit vectors and simplify their expression (e.g. Isabelle was used to go from VCs looking like Listing 4 towards the pre in Listing 3). Once a manageable size was reached, the actual specification within the complicated expression became clear and enabled the discharge of the VCs.

As a result of the analysis the development team provided additional test data and simulations to ensure that the potential problems could not occur in practice. Overall, the experience was positive and produced results in identifying potentially serious issues, that were easily mitigating with more targeted testing (i.e., tests based on formal specifications). Finally, it was also possible to establish that the code being submitted for certification was close to MISRA compliance (that is all mandatory and most recommended and advisory rule violations).

10. Related work

The formal analysis methods described in this work are not novel. Similar techniques were being described and applied in the 1990s. For example, a mature set of tools have been developed by Heitmeyer's team using SCR [11]. Their approach uses a tabular notation to describe requirements which makes the technique relatively acceptable to developers. Atlee and Gannon described a similar approach in [19]. In some domains, other than medical domains, formal mathematically based methods have been effective in analysing and assessing risks systematically (see for example, [20, 21]).

Despite the success of these techniques there is a continuing perception that formal methods are not easy to use and that they cannot be scaled to substantial systems. These barriers to their use have limited their uptake in medical domains. Recent research with the cooperation of the US Food and Drug Administration (FDA) have led to increased possibilities for their potential use [22, 12].

Other work considers the analysis of requirements that are initially formalised in terms of an architectural model. It is clear that this kind of analysis is not limited to systems that can be reduced to a control table such as the one that the analysts were provide with in this case. The work of Mavaido and others for example [23] develops a set of properties that are guided by the architectural style of the system under analysis thus achieving correctness by construction.

There are of course several ways in which it can be demonstrated that a device satisfies safety requirements using formal techniques when formal methods are used as part of the design process at the early stages of development. Tools such as Event B [24, 25] enable the development of an initial model that specifies the device characteristics and incorporates the safety requirements. This initial model is gradually refined using details about how specific functionalities are implemented. This was not a realistic approach in the present case because, when the analysis was to be done, the device had already been developed. Indeed such techniques are currently not feasible given the typical resources available to medical device developers. There were also many unknowns at the early stages of NIDUS so that an agile approach to software development was essential.

Alternatively a model could be generated from the program code of an existing device, using a set of transformation rules that guarantee correctness, as discussed in [26]. This approach could have been used for other software aspects of the device, however it is unclear how well such techniques scale. Proving the equivalence of model with the software component in relation to the device was not a problem for the particular example because the software was driven by a table and the table was translated directly into the model. Some source analysis has been performed but this does not prove that the software drivers themselves were implemented correctly. However, the hardware drivers are mostly very simple, at the level of 'open valve', 'close valve', 'stop', 'go slow' and 'go fast'. Software here is easy to verify either formally or using traditional test methods.

Finally, a key contribution of this paper is that it illustrates the potential

for collaboration between a small team of developers and a formal analysis to aid the process of producing a risk analysis for a system. In this case the risk log was initially developed based on previous experience and the designer's understanding of the novel implications of the new design. Other work describes collaborations between domain experts and software developers in other contexts [27] and the role of various formal techniques to develop a system [28, 29]. These approaches typically assume a development team that harnesses these techniques concurrently from the outset.

11. Discussion

The practical use of formal techniques, as part of the risk analysis of a medical device within a small organisation, remains a prospect rather than a reality. The contribution of this paper is a practical demonstration of the use of formal techniques to analyse a component of a safety critical system. The novelty in this paper has been to apply this technique within a team where typically small teams with limited resources are involved.

The size of the translation of the control table is already discussed in Section 5. The development of the first model, by hand, took about seven hours. It was possible to make most changes to the model and show the results interactively during meetings with the development team without disturbing the flow of the meeting. Hence the refinement of requirements and the careful analysis of the hazards were facilitated by the process. As discussed in Section 7, a set of 252 requirements were identified in the risk log of which 47 mitigations used the MAL analysis at least in part. The analysis involved 23 properties. These supported mitigations relating both to aspects of protection and design. On the rare occasions when it was not possible to refine a property during the meeting, for example when meta-attributes were required, this could be achieved within an hour outside the meeting. Verifying all the properties together on a MacBook Pro with Intel Core i5 clocked at 2.9GHz, with 8GB RAM and SSD memory, took 1.7 seconds. The exercise shows that, with appropriate expertise and using available artefacts (the table, safety requirements), the use of formal methods required little additional effort and supported effective discussion of the risks between the developers. Applying the formal tools is not *simply* a matter of proving the system right or wrong, but of focusing the discussion and the overall analysis effort on those aspects of the system that are most critical.

Overall, we found the process of identifying, adapting and applying formal techniques to the medical device domain promising. On the one hand, various hardware design principles applied to the "in-house" software development process were already quite strict and of high-quality. On the other hand, certain aspects of the design and code could benefit from key (and simple) formal methods principles, such as model checking the state transition design, and performing design-by-contract with MISRA-C++ compliance. Testimony to this is that we did not encounter any major/serious flaws in our investigations, but rather potential sources of future problems, as well as guarantees that the design satisfied the safety properties of interest. The exercise of discovering and applying suitable formal techniques to the dialyser was positive as it served to identify the needs of the engineers and the limitations of their processes. For example, writing certain contracts, in particular refinement ones, is not something developers would like to do themselves; on the other hand, there are other contracts developers would be willing to write themselves. It also found a couple of potentially serious bugs that could be a problem if the code is to be ported to a different target.

A key difficulty faced with the source code analysis was to find a tool that would handle a mixture of C and C++. Since only limited changes to the code were possible, Microsoft VCC [2], Frama-C [3], or Verifast [4] were all considered. These tools focus on either concurrent C and/or memory safety with respect to pointer usage. Given that the software code was a mix of C and C++, embedded code makes little use of dynamic memory, and that the dialyser was sequential, these tools were not suitable. Moreover, these other tools do not check for MISRA compliance, a strengthening argument that can be used in the certification process.

The risk analysis process described in the paper succeeded because the software controller of the device was driven by a table and it was relatively easy to generate a model from the table. It also succeeded because a mixed disciplinary team was involved. This team included one person who was able to use the formal tools and provide an explanation of the requirements and model formulations. It is standard practice to use a table to drive software that controls a multi-step process as in this case. However there are cases

[2]See `http://research.microsoft.com/en-us/projects/vcc/`.

[3]See `http://frama-c.com/`.

[4]See `www.cs.kuleuven.be/~bartj/verifast/`.

where this does not happen and moreover, as in this case, the software covered by the controller is only part of the software. As noted in the previous section other architectural models facilitate the development of similar analyses [23].

The dialysis machine also includes user interface features, for example capacity to enter new values for thresholds relevant to the dialysis process. These are involved in the initial set-up of the machine. Other analyses, involving several of the authors, have focussed on existing intravenous infusion pumps [30]. In these cases, there were few sequences such as the dialysis cycle and a table driven process was less relevant. The analysis described in this paper therefore raises questions about the potential for extending this approach to a broader class of medical systems. The challenges raised by this analysis in the context of small-scale developments, such as this one, are:

- *Systematic modelling:* While formal approaches to the development of software that refine safety requirements exist (see [31]), these are not yet feasible to use given the available tools and skills of existing small development teams. In this case the formal methods expertise was recruited short-term for the purpose.
- *Mixed disciplinary teams:* There was substantial benefit in recognising and using expertise from sources outside the development team. A mixed discipline approach is already in practice in the case of small companies or innovative pre-commercial developments. It would make sense therefore to add support for these analytical skills to the toolkit to enable device developers to use them.
- *Mixed styles of analysis:* As in this case, a well defined and yet important software component may be analysed formally. The formal analysis of the controller table can also improve the testing coverage of the device drivers themselves although this was not done in this case. It is also good practice to have multiple independent arguments to demonstrate the safety of the system. Hence it makes sound sense to use formal techniques to improve confidence in the risk analysis.

12. Conclusion

This paper illustrated how formal techniques may be used successfully as part of the risk analysis process associated with the development of a medical device. The analysis described in this work is part of the documentation created for a device submission that has gone forward for regulation. The safety requirements that were formulated and proved, and improvements

when a property that was formulated to reflect a requirement failed to be true, illustrate how the analysis led to improvement in the safety of the design while providing a concise basis for evidence that part of the system is safe. The technique is readily repeatable. Tools that have been developed allow the automated development of models from control tables. The analysis approach complements testing techniques and provides a systematic solution to the safety assessment of critical devices.

Acknowledgements. This work has been funded by: EPSRC research grant EP/G059063/1: CHI+MED (Computer–Human Interaction for Medical Devices); and NanoSTIMA (ref. NORTE-01-0145-FEDER-000016) financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). Leo Freitas would like to acknowledge EPSRC Trams2 project for financial support, Andrew Sims for providing access to the dialyser, which was used as our case study and Aleksands Baklanovs for doing some of the source analysis as part of an undergraduate project

13. References

[1] BSI, Medical device software - software life cycle processes, Tech. Rep. BS EN 62304:2006, British Standards Institution, CENELEC, Avenue Marnix 17, B-1000 Brussels (2008).

[2] US Food and Drug Administration, General principles of software validation: Final guidance for industry and FDA staff, Tech. rep., Center for Devices and Radiological Health, available at `http://http://www.fda.gov/medicaldevices/deviceregulationandguidance` (January 2002).

[3] M. G. Coulthard, J. Crosier, C. Griffiths, J. Smith, M. Drinnan, M. Whitaker, R. Beckwith, J. N. S. Matthews, P. Flecknell, H. J. Lambert, Haemodialysing babies weighing < 8kg with the newcastle infant dialysis and ultrafiltration system (NIDUS): comparison with peritoneal and conventional haemodialysis, Pediatric Nephrology 29 (10) (2014) 1873–1881. doi:10.1007/s00467-014-2923-3.
URL `https://doi.org/10.1007/s00467-014-2923-3`

[4] J. C. Campos, M. D. Harrison, Systematic analysis of control panel interfaces using formal tools, in: N. Graham, P. Palanque (Eds.), Interactive systems: Design, Specification and Verification, DSVIS '08, no.

5136 in Lecture Notes in Computer Science, Springer-Verlag, 2008, pp. 72–85.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: An Open Source Tool for Symbolic Model Checking, in: K. G. Larsen, E. Brinksma (Eds.), Computer-Aided Verification (CAV '02), Vol. 2404 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 359–364.

[6] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, MIT Press, 1999.

[7] C. Tapp, An introduction to MISRA C++, SAE Int. J. Passeng. Cars - Electron. Electr. Syst. 1 (1) (2009) 265–268.

[8] J. Carlton, D. Crocker, Escher Verification Studio Perfect Developer and Escher C Verifier, J. Wiley and Sons, 2012, Ch. 5, pp. 155–193.

[9] M. Ryan, J. Fiadeiro, T. Maibaum, Sharing actions and attributes in modal action logic, in: Theoretical Aspects of Computer Software, Vol. 526 of Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 569–593.

[10] A. Monk, M. Curry, P. Wright, Why industry doesn't use the wonderful notations we researchers have given them to reason about their designs, in: D. Gilmore, R. Winder, F. Detienne (Eds.), User-centred requirements for software engineering, Springer, 1991, pp. 185–189.

[11] C. Heitmeyer, J. Kirby, B. Labaw, R. Bharadwaj, SCR: A toolset for specifying and analyzing software requirements, in: Computer Aided Verification, Springer-Verlag, 1998, pp. 526–531.

[12] P. Masci, A. Ayoub, P. Curzon, M. Harrison, I. Lee, O. Sokolsky, H. Thimbleby, Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example, in: Proceedings ACM Symposium Engineering Interactive Systems (EICS 2013), ACM Press, 2013, pp. 81–90.

[13] M. Dwyer, G. Avrunin, J. Corbett, Property Specification Patterns for Finite-State Verification, in: M. Ardis (Ed.), 2nd Workshop on Formal Methods in Software Practice, 1998, pp. 7–15.

[14] S. Vadera, F. Meziane, From english to formal specifications, The Computer Journal 37 (9) (1994) 753–763.

[15] D. Crocker, Can C++ be made as safe as SPARK?, in: ACM Proceedings High Integrity Language Technology, 2014.

[16] D. Crocker, Perfect developer reference manual v6.10, Tech. rep., Escher Technologies Ltd. (December 2013).

[17] L. Freitas, A. Stabler, Translation strategies for medical device control software, Tech. rep., Newcastle University (August 2015).

[18] T. Nipkow, L. Paulson, M. Wenzel, Isabelle/HOL: a proof assistant for Higher-Order Logic, no. 2283 in Lecture Notes in Computer Science, Springer-Verlag, 2002.

[19] J. M. Atlee, J. Gannon, State-based model checking of event-driven system requirements, IEEE Transactions on Software Engineering 19 (1) (1993) 24–40.

[20] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, G. Heiser, Comprehensive formal verification of an OS microkernel, ACM Trans. Comput. Syst. 32 (1) (2014) 2.
URL http://doi.acm.org/10.1145/2560537

[21] J. Barnes, R. Chapman, R. Johnson, B. Everett, D. Cooper, Engineering the tokeneer enclave protection software, in: IEEE International Symposium on Secure Software Engineering, IEEE, 2006.

[22] B. Kim, A. Ayoub, O. Sokolsky, I. Lee, P. Jones, Y. Zhang, R. Jetley, Safety-assured development of the GPCA infusion pump software, in: Proceedings of the ninth ACM international conference on Embedded software, EMSOFT '11, ACM, New York, NY, USA, 2011, pp. 155–164.
URL http://doi.acm.org/10.1145/2038642.2038667

[23] A. Mavridou, E. Stachtiari, S. Bliudze, A. Ivanov, P. Katsaros, J. Sifakis, Architecture-based design: A satellite on-board software case study, in: O. Kouchnarenko, R. Khosravi (Eds.), Formal Aspects of Component Software, Springer International Publishing, Cham, 2017, pp. 260–279.

[24] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, 2010.

[25] R. Banach, Hemodialysis machine in hybrid Event-B, in: M. Butler, K.-D. Schewe, A. Mashkoor, M. Biro (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z, Springer International Publishing, Cham, 2016, pp. 376–393.

[26] G. J. Holzmann, Trends in software verification, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), FME 2003: Formal Methods, Vol. 2805 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 40–50.

[27] D. Coppit, K. Sullivan, Formal specification in collaborative design of critical software tools, in: Proc. 3rd. IEEE International High Assurance Systems Engineering Symposium, IEEE Computer Society Press, 1998, pp. 13–30.

[28] G. Garcia, X. Roser, Enhancing integrated design modelbased process and engineering tool environment: Towards an integration of functional analysis, operational analysis and knowledge capitalisation into co-engineering practices, Concurrent Engineering 26 (1) (2018) 43–54. doi:10.1177/1063293X17737357.

[29] J. S. Fitzgerald, P. G. Larsen, K. G. Pierce, M. H. G. Verhoef, A formal approach to collaborative modelling and co-simulation for embedded systems, Mathematical Structures in Computer Science 23 (4) (2013) 726750. doi:10.1017/S0960129512000242.

[30] M. D. Harrison, P. Masci, J. C. Campos, P. Curzon, Verification of user interface software: the example of use-related safety requirements and programmable medical devices, IEEE Transactions on Human Machine Systems 47 (6) (2017) 834–846. doi:10.1109/THMS.2017.2717910.

[31] S. Yeganefard, M. Butler, Structuring functional requirements of control systems to facilitate refinement-based formalisation, in: Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011), Vol. 46, Electronic Communications of the EASST, 2011.