From Informal System Requirements to Formal Software Specifications - An Experience Report*

Anitha Murugesan, Dan Cofer, Michael Whalen, and Mats Heimdahl

Department of Computer Science and Engineering,
University of Minnesota, 200 Union Street, Minneapolis, MN 55455,USA
{muru0011,cofer008, whal0046, heimd002}@umn.edu

Abstract. The benefits of formal techniques at their intersection with requirements engineering is tricky to access. Formal methods have been enormously useful in verifying complex system requirements. However, their success depends on precisely formalizing "what" needs to be verified and thoroughly understanding "how" it is verified. While the advances in this field has given rise to sophisticated techniques and tools, there is a lack of awareness and guidance in appropriately using them, that often makes their use painful and results misleading.
In this paper, we report on the challenges and non-obvious nuances in using formal methods to verify the requirements of hierarchically developed complex systems that, we believe, are recurrent concerns yet inadequately addressed in the formal requirements analysis community. In our experience, we found that (a) precisely identifying the contextual information for requirements when formalizing requirements from traditionally structured requirements document is a non-trivial task, (b) failure to recognize the subtility between requirements when formalizing them at various levels of abstraction leads to believable, but highly misleading verification, (c) lapse in detecting the fallibility of the tools could leads to incorrect proofs about systems, and (d) inadequate mitigation of risks associated with different tools usage could lead to misplaced confidence about the system. In the sequel, we explain our approach to identify, mitigate and address such concerns. We believe that these lessons learned would prove instructional for practitioners involved in similar efforts.

1 Introduction

Complex cyber-physical systems are often developed hierarchically through decomposition of a "top level" system into layers of smaller and more manageable components. Decomposition is performed to accommodate reuse, integration with existing or commercially developed components, and distributed development, as well as to factor the complexity of development [**?**], and the process of decomposition typically involves identifying components and their interconnections as well as allocating requirements to each of them. One of the challenging tasks in this process is determining *"if the composition of component behaviours satisfy the overall system requirements?"*. In safety critical systems, formal methods (mathematical techniques) can be used to rigorously

* This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

ensure the correctness of systems. However, adopting formal techniques alone does not guarantee the correctness of the system.

The challenge in successfully using formal methods lies in precisely stating "what" needs to be verified and thoroughly understanding "how" it is verified, rather than the action of verification itself. Nowadays, sophisticated techniques and tools can help automatically verify the correctness of models with respect to some properties of interest. But these tools rely on the skill level of the engineers to *formalize* the requirements and understand the details of the tools used. For example, using formal tools one could successfully prove a set of logically inconsistent or vacuous requirements and be happy about the successful but meaningless verification. While, there are numerous success stories that illustrate the benefits of using formal methods [**?**, **?**] as well as discussion about the limitations of formal methods [**?**, **?**], there are not adequate discussions about the practical challenges encountered and techniques adopted to mitigate them while using formal methods in real applications. We believe that, sharing such experiences will allow the wider dissemination of formal methods in industrial use.

In previous work [**?**, **?**] we proposed scalable and efficient model based approaches to verify the formalized requirements of complex control systems, given its architecture (hierarchical decomposition into components) and the formalized requirements. The approach involves hierarchically modeling, formalizing and verifying requirements of system and its components using multiple tools (Simulink, Simulink Design Verifier [**?**] and AADL/AGREE [**?**]), each with slightly different formal representations and usage. While this approach was undoubtedly useful to verify large complex systems effectively, when we used it to verify a industry sized infusion pump system, a medical device, we faced a number of practical challenges that had the potential to undermine the value of not just our approach but formal requirements analysis in general. To the best of our knowledge these challenges, that lie in the intersection of the requirements engineering and formal verification domains, have not been discussed in depth and addressed in any existing literature.

In this paper we describe those practical challenges we faced in the process of constructing models and formalizing requirements at multiple levels of hierarchy when applying tools "at scale" involving several developers over an extended period of time. We focus on four challenges:

Requirements Formalization: Systematically identifying the contextual information while formalizing the requirements from traditionally documented natural language requirements is a time consuming task. In our earlier efforts to formalize the requirements of control systems, although they were documented using standard patterns [**?**] and templates, we spent a number of hours to precisely formalize it, i.e to identify the precise antecedent/ precondition. To overcome this challenge, we propose a hierarchical structuring scheme to document requirements, that we believe, not only clarifies the context for each requirement while formalizing them but also naturally reflects how requirements are conceived by the stakeholders in the critical system domain.

Requirements Refinement: We initially mapped system requirements to software requirements as is advocated in [**?**] (and used in other papers such as [**?**, **?**]), but found

that applying this approach led to incorrect compositions when we tried to prove system requirements.

Tool Fallibilities: In an architectural proof approach, all proofs are predicated on "leaf-level" components meeting their specification. If these leaf-level specifications are incorrect (for example, it is not possible to construct an implementation meeting the specification or it is inconsistent after a certain number of time steps of execution), then the proof is not well-founded. This has forced us to develop tools towards checking *realizability* and strengthen the existing *consistency checking* of requirements.

Matching Tool Boundaries For scalability concerns, we had to move verification results from one tool to another. However, the mere action of transcription of requirements between different notations of the tools (even through the tools shared the "same" semantics) induced several errors that lead to misplaced confidence in results. This motivated us to implement tools and rigours practices to systematically check the translation.

The contribution of the paper is reporting the challenges and pitfalls that engineers typically encounter while formally reasoning about complex control system requirements, especially in the safety critical system domain. We also describe the solutions that we adopted to overcome those challenges. While we illustrate these lessons learnt using the infusion pump as a case example, we believe many engineers working on similar applications will face similar challenges, so we hope that sharing our experiences prove instructive.

The paper is organized as follows. In Section 2 we provide a brief overview of our previous effort that serves as a context for the issues described later. In Section 3 we explain the specific challenges and pitfalls we encountered while verifying the infusion pump system. In Section **??**, we elaborate our approachs and recommendations to address the challenges. Finally, we conclude the paper in Section **??**.

2 Background

In this section, we describe our prior work [**?**, **?**] on modeling and formally verifying the requirements of an infusion pump that forms the basis of this paper. We provide a brief overview of an infusion pump system, its modeling and verification activities that is required to understand the formalization challenges described later in the paper. Infusion pumps are medical cyber physical systems used for controlled delivery of liquid drugs into a patient's body according to a physician's prescription (the set of instructions that governs infusion rates for a medication). Unfortunately, these devices have been involved in many hazardous incidents that triggered the need to strengthen the current system development practices for such systems. Our aim is to demonstrate a rigours model based development approach using a generic infusion pump system that, along with the artifacts we develop in the process, can be be used as a reference for researchers, manufacturers and certification authorities.

2.1 System Overview

We considered a Generic Patient Controlled Analgesia Infusion Pump (GPCA), a special type of infusion pump, is typically equipped with a feature that allows patients to self-administer a controlled amount of drug (a patient-bolus), typically a pain medication. Figure 1 shows a GPCA device in a typical usage environment in which a clinician sets up the device and the patient receives the medication from the device through an intravenous needle according to the programmed prescription. To enhance safety of the device it is usually connected to a repository (hospital pharmacy database) that stores manufacturer provided drug limits. In order to analyse the problems associated with

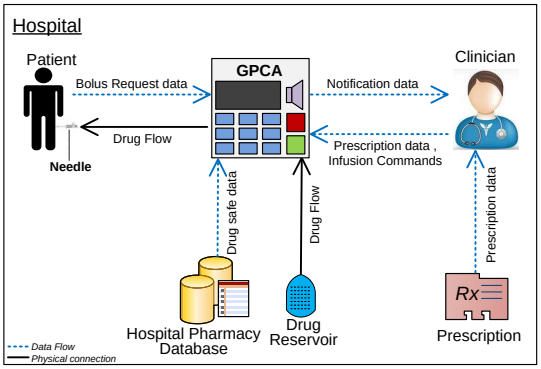

Fig. 1: GPCA System Overview

modern infusion pumps available in the market, we considered several advanced capabilities in the GPCA. For instance, we considered three types of infusion options for drug delivery for the GPCA - (a) *basal* infusion in which the drug is delivered at a constant (and usually low) rate for an extended period of time, (b) *intermittent bolus* infusion that delivers drug at a higher rate for a short duration at prescribed time intervals according to some therapy regimen, and (c) *patient bolus* that delivers additional drug in response to a patient's request for more medication. In addition, we also considered advanced hazard mitigation capabilities that detects hazardous anomalies/behaviours and mitigates the hazard by notifying the clinician and/or inhibiting infusion and other device operations.

2.2 System Modeling and Verification

Our interest with the GPCA is to develop and demonstrate a rigours end-to-end development approach that along with the artifacts developed could be used as a reference for manufactures to develop devices assured safety and efficacy. We initially captured a reasonably complete set of overall system requirements using documentation patterns and templates that we initially thought will make the modeling and requirements formalization easy. Later to analyse the behaviors of a system that satisfied the requirements we

developed a behavioral model of the GPCA. However, to cope with the complexity of modeling the behaviours of the system we had to hierarchically decompose it into various components. This induced the need to identify precise requirements for each component and the architecture (components and its interconnections) and analyse if the decomposition satisfies the system requirements. To automate the verification process, we used a number of notations and tools in this approach. First, we used Architectural Analysis & Design Language (AADL) to capture the architecture of the GPCA (components and its interconnections). To capture the component allocated requirements we used an extension of the AADL language that supports specification of formal textual requirements along with its respective components (and system) in the AADL model. To verify the correctness of the decomposition we used a compositional reasoning tool named AGREE [**?**]. Given the system architecture and the allocated requirements (captured as assumptions and guarantees) for both the system and its components, AGREE automatically verifies if the system level guarantees holds as a logical consequence of both component contracts and system level assumptions. While AGREE helped verify the correctness of GPCA decomposition, it assumes the correctness of the leaf level component requirements. Hence, to verify those requirements we modeled the behavior of each leaf level component using Simulink/Stateflow tools and verified them using MathWorks Simulink Design Verifier tool. The traceability between the leaf level requirements in AGREE and Simulink Design Verifier was straightforward since the notations used by both the tools were semantically same and we manually documented the trace. This approach and tool chain allowed us to rigorously demonstrate an end-to-end large scale system verification using tools and notations commonly used in the industry.

3 Requirements Engineering Challenges

Although we successfully demonstrated the approach using the GPCA, our journey from GPCA's natural language requirements to the fully verified formalized system, component and sub-component requirements was not easy. We faced a number of challenges that we believe is typically encountered by engineering performing similar tasks. In this section we describe those challenges in detail.

3.1 Challenge 1: Requirements Formalization

While model-based approaches helps verify requirements, the process of formalizing such requirements from the natural language documents is often a laborious process. One of the major reasons for this challenge comes from the fact that natural language depends heavily on context. To make a smooth transition from the natural language to the formal specification languages we need to systematically identify the contextual information for each requirement. While tools such as model checkers return counter examples to help engineers discover the contextual information, the counter example directed context exploration is a time-consuming task especially for certain types of requirements.

While formalizing the requirements of GPCA, we found that systematically identifying the context for two groups of requirements was challenging. The first set of

requirements that we had trouble formalizing were those that describe the behaviour of the system under under normal working conditions. For example, one of the requirements states[1] that,

> *"When the patient requests a bolus, the system shall deliver an the drug at flow rate equal to* $patient\_flow\_rate$ *"*

When we tried to formalize the requirement as is and verify it, the model checker repeatedly returned counter examples. A careful examination of the counter examples revealed that there were exceptional conditions in the system that prevented the infusion to occur and this requirement was not satisfied in that context. Those conditions were actually safety features of the system to prevent hazards were documented as "alarm requirements" in another section in the requirements document. Unfortunately, establishing and maintaining traceability between such requirements was not practically easy, due to the numerous requirements and the orthogonality/exclusivity between the behaviours they describe.

The second group of requirements that we had trouble formalizing were those that were mutually exclusive with each other but with a certain inherent priority among them. In the GPCA, the alarming requirements capture the system's responses to exceptional conditions. There were 18 exceptional conditions identified for the GPCA, each with its own set of desired system responses depending upon the severity level of that condition. For example, if the drug reservoir is empty, one of the highest severity condition, the system shall raise audio alarm, display error message and stop infusion. On the other hand, when the system is idle for a long time, a low severity level condition, the system only displays an appropriate message. One of the problems we encountered was formalizing the requirement in such a way that verifies their effect independently. For instance, to verify a lower priority condition requirement we had to systematically ensure all the higher priority ones did not occur. The concern is that there was no explicit priority that was specified in the requirements document and we had to infer the priority based on the system responses. Unfortunately, there was no guidance or patterns to help systematically organize and formalize such requirements for verification. On the contrary, the GPCA model that implements the requirement had a specific prioritization mechanism (that we believe is a design decision of the developer implementing the requirements). Formalizing and verifying each alarm condition requirements independently without including the design decisions of the model was a big challenge. We realized that root cause of this problem is due to undisciplined manner in which the requirements were organized in the document.

3.2 Challenge 2: Requirements Refinement

When systems are decomposed into components, deriving the component requirements from the system requirements is a challenging task. While tools like AGREE help us to

[1] We intensionally simplified this requirement such that it illustrates the problem. However, the original requirement had more conditions associated with it.

verify if component requirements are sufficient to guarantee the system level requirement, deriving those component requirements and formalizing them for verification has largely been a manual and error prone activity. Unfortunately, there is little/no guidance to help systematically identify component requirements.

In an effort to provide guidance to engineers performing and formally verifying the system decomposition, Miller et al [**?**] propose that that most system requirements can be recaptured into identical software requirements by minor modification to the scope. Their approach is based on the famous four variable model that formally captures the high level artifacts of a typical process control system, the requirements (REQ), the sensor functions(IN), controller functions(SOF), actuator functions (OUT) and their environment (NAT). The main contribution of the four variable model is mathematically defining the artifacts, their scope (the four variables - **mon**itored, **con**trolled, **in**put and **out**put used to express the artifacts) and their inter-relationships required to reason about their correctness. However, the four variable model doesn't clarify how to derive and specify the functions of the components.

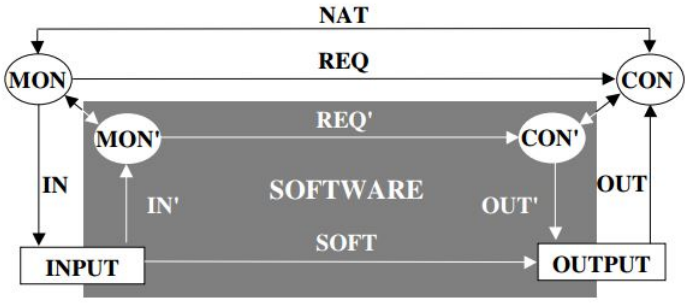

Fig. 2: Extension of Four Variable Model

To address this concern for the software, Miller et al extended the four variable model, as shown in Figure 2. They recreate virtual versions of the variables mon' and con' that differ from the original mon and con in terms of value and timing introduced when sensing and setting the input and output variables. Using these virtual variables, they "stretch" the SOF (software) relationship into IN', REQ', and OUT'. The IN' and OUT' represents the specification of hardware drivers that were previously part of the SOF. With this change, they propose a mere recapture of each function in REQ to REQ' using corresponding variables. They assert that this makes the tracing of the subsystem requirements REQ to the software (REQ') direct and straightforward. While this approach superficially seems indisputable and makes the task of decomposition simpler, in our experience, we found that this approach could result in inaccurately specified and verified component requirements.

The overall system requirements especially for complex control systems such as GPCA are typically captured in such a way to accommodate certain degree of inaccuracies and imperfections. For example, lets consider a system level requirement:

"In basal mode, the system shall infuse the drug at a flow rate within $BasalFr \pm t\%$ tolerance"

The tolerance (t) in this requirement was included to accommodate inaccuracies of the physical components that is going to be used in the GPCA. Mathematically, this requirement is a relation, since the flow rate (output) is allowed to have range of values. When we decomposed the GPCA into various components and tried to allocate requirements to the software (one of the components of GPCA), we were tempted to formalize requirements for verification such that it mirrors the system requirement but in terms of inputs and outputs of the software, such as:

```
(Mode = basal) ⇒
      FlowRate ≤ (1 + t/100) * (BasalFr) and
      FlowRate ≥ (1 - t/100) * (BasalFr)
```

While the above formulation appears to be correct and mirrors the system level requirement, the tolerance part was not intended to be allocated to the software. When we model checked this requirement it was successfully verified since the software indeed satisfied, but in an unintended fashion. In reality, the software's output is deterministic and there was no need for the tolerance, hence the consequent in the formalization was supposed to be `(FlowRate = BasalFr)`. That is, mathematically the requirement of the software is a function as opposed to the system level requirement that is a relation.

While, superficially this doesn't appear to be a problem, an indepth analysis revels that this leads to a situation in which we could prove the system level requirements with just the software requirements. The way we formalized the software requirement is not only an over approximation of the capabilities of the software but also masks the other component requirements. Unfortunately, this was not apparently visible until we started analysing how the software requirements realizes the system level requirements.

3.3 Tool Fallibilities

Although we used sophisticated tools with advanced capabilities such as consistency checks, cursory knowledge about how the tools perform such checks lead us to successfully verify inconsistent requirements. The tool we used, AGREE, had capabilities such as consistency check in addition to verification of requirements. The consistency check features detects logical contradictions among requirements. The intent of performing such checks is to identify if the verification was trivially successful due to presence of such inconsistent requirements. While all the GPCA software requirements were successful, our tool expert, during the manually inspection phase, found that there were several self-inconsistent requirements. Surprising those inconsistencies were not identified by the tool. This was primarily because we did not have full knowledge of the tools setting that associated with that check.

Lets us illustrate the problem with an concrete example. To formalize certain requirements for GPCA we had to specify counters that could keep track of duration

of internal conditions or occurrence of certain actions. For example, consider a requirement to notify the clinician when the patient requests more than a certain number of boluses. To formalize this requirement we had to count the number of requests received (PatientRequest) by the system. Hence, we declared an integer variable (boluscnt), whose initial value was 0 and then gets incremented whenever a PatientRequest is received by the system, as shown below. In AGREE, "-¿" is used to capture the initial and subsequent value computations for a variable.

```
boluscnt:int = 0 -> boluscnt + PatientRequest
```

While the tool successfully verified the requirement involving the boluscnt variable, during manual inspection our tool expert pointed out this statement was internally inconsistent. The inconsistency was due to the usage of boluscnt in both the right and left side of the equation. According to the tool, the value for boluscnt always refers to its value in the current time step, whereas we actually intended to refer to the value in the previous time step in the right side of the expression. The incorrect formulation had caused a circular dependency in assigning values for boluscnt variable, that made this expression and the requirement involving this variable inconsistent and trivially satisfied. In the GPCA, we identified 10 such inconsistent formalizations. To address this issue, we used AGREE's operator *"pre"*, that refers to the value of a variable in the previous step as shown below.

```
boluscnt:int = 0 -> pre(boluscnt) + PatientRequest
```

While adding *"pre"* fixed this particular requirement's inconsistency, our concern was that the tool's consistency checker did not identify this problem. After discussing with the tool developers, we found that the tool was set up by default to check for consistency only in the initial time step (first step of execution). This was not a bug in the tool, rather an intensional default setting to optimize the performance of the tool. According to the tool developers, most inconsistencies can be found in the first step and this was an exceptional condition. Since the boluscnt equation was inconsistent only after the initial step, the consistency checker did not report the problem.

3.4 Matching Tool Boundaries

While it is ideal to use a tool/technique to verify the entire system's requirements, in practice it is inevitable to avoid the use of multiple tools for scalability concerns and different verification needs. One of the well known problems associated with using multiple tools is matching the sematic differences between them. However, even when we use tools that are have same semantics, errors arise when recapturing requirements between the tools.

In GPCA, we used a combination of two tools to cope with scalability of verification. We used AGREE to model the system architecture and verify the decomposition of system requirements into component requirements. Subsequently, we also developed

detailed behavioral models for each component in Simulink and verified if the component requirements verified in AGREE are indeed satisfied by the respective component's behavioral model using Simulink Design Verifier (SDV). The advantage of this tool combination in addition to being scalable is that the semantics of the notation used in AGREE and SDV - Embedded Matlab - to specify requirements are identical. Hence, we presumed that the translation of requirements between AGREE and Simulink notations can be done in a error free manner.

While a majority of the requirements were correctly recaptured between the notations some requirements were incorrectly recaptured that, unfortunately, was not easy to detect. We identified two issues. One was transcription errors and the other was requirement management issues. The transcription errors occurred in the process of manually recapturing requirements. When recapturing a few requirements we unknowingly changed the operators. In GPCA, we translated the following requirement that was proved in Simulink,

`SystemOn and (AlarmLevel = Med)` $\Rightarrow$ `FlowRate` $\leq$ `LowFlowRate`

into AGREE, as follows:

`SystemOn and (AlarmLevel = Med)` $\Rightarrow$ `Flowrate` $<$ `LowFlowRate`

It was not possible to dismiss this error as a mere typing error, since it demonstrates the potential for occurrence of such issues. Unfortunately, this error was not visible until we checked for AGREE's feature to verify realizability or feasibility of requirements. When AGREE tool returned that the above requirement was not realizable, we were surprised since it was proven in the detailed behavioral model in Simulink. But a careful inspection of the requirements reveled that there was a transcription error and we manually corrected 4 such errors in the requirements.

Another issue, that we believe is more common, is maintaining the synchrony between the requirements in both tools. After we proved requirements in AGREE, when we recaptured it for verification using SDV, at times we had to slightly change the way requirements were formalized, such that it is verifiable in the behavioral Simulink. However, such changes were not always corrected and reverified in AGREE. This caused mismatch between the requirements in AGREE and Simulink.

Although we were diligent in recapturing the requirements between the tools, human errors and negligence was unavoidable. For the GPCA, we did additional manual inspections to verify the synchrony. However, our long term goal is to fully automate the translation and check for synchrony to avoid such issues. Infact, our recommendation to engineers performing such tasks is to ensure there is either a strict process or automation to avoid such issues.

4 Recommendations

4.1 Solution To Challenge 1: Structuring Requirements

Instead of following the trial and error process to refine formalized requirements in ways to verify that requirement, we tried to address this issue at its root cause - the requirement document. The system behaviors of control systems are typically grouped in terms of *modes* - a logical way to describe a set of system behaviours. In a complex system, there could be many such modes and they could either be mutually exclusive (only one mode can be active at a time) or orthogonal (multiple modes can be active at a time). Lucent understanding of such modes and their groups was crucial to systematically formalize the requirements. In the GPCA, there were three main orthogonal groups of modes - infusion, alarm and configuration. Within each orthogonal group, there were a set of modes that were mutually exclusive of each other. For instance, within infusion mode group basal, intermittent and patient bolus modes are mutually exclusive to each other. Similarly, in the alarm group, each each exceptional condition was considered mutually exclusive to each other. We decided to leverage this pattern of modes and use it to organize requirement statements in the the requirements document.

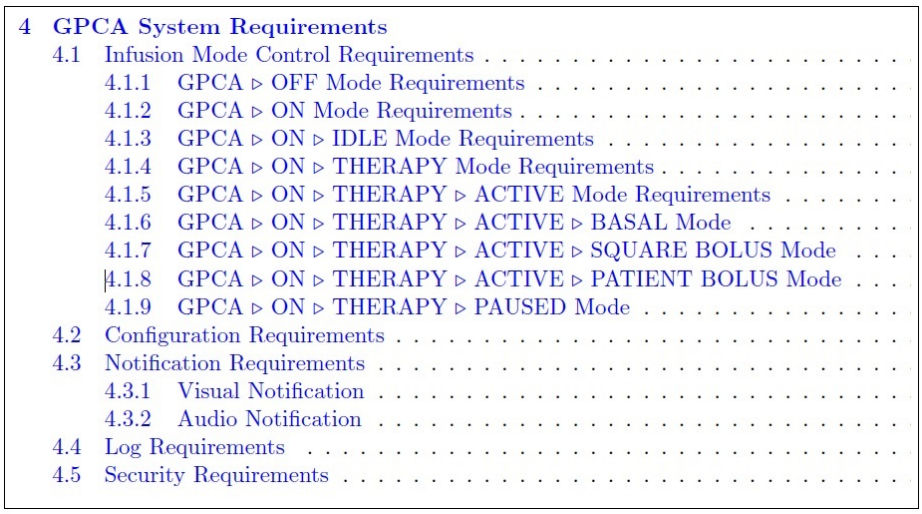

Fig. 3: GPCA System Requirements Structuring

We restructured the requirement document and followed a hierarchical structure to organize requirements of the GPCA, as shown in Figure 3 that not only simplified the formalization process, but also brought about conceptually clarity of the requirements. In this structure, the requirements that were common to the entire system, irrespective of the modes, such as startup behavioural requirements were placed at the top of the hierarchy. At the second level in the hierarchy, we placed the three mode groups - infusion, alarm and configuration - that indicated the orthogonality among them. Within

each orthogonal mode group, we again followed the hierarchical structure to organize their requirements. The infusion mode group had three mutually exclusive modes - basal, intermittent bolus and patient bolus. While documenting their requirements, we observed that there were only a few requirements that were specific to each of those infusion modes whereas several requirements were common to all of them. To avoid repetition of requirements, we grouped all the common requirements to a parent mode called "therapy", and then organized the three infusion modes as its child modes with specific requirements.

Condition	Severity Level	Notification		Inhibit All Infusion	Inhibit Bolus	
		Visual	Audio		Intermittent	Patient
Empty Reservoir	High	Yes	Yes	Yes	Yes	Yes
Over Infusion						
Environmental Errors						
Device Errors						
Air embolism	Medium		Yes	Allow only KVO	Yes	Yes
Occlusion						
Door Closed						
Low Reservoir	Low		No	No	No	Yes
Under Infusion						
Idle Time Exceeded	Warning		No	No	No	No
Paused Time Exceeded						
Config Time Exceeded						
Pump Temperature						
Battery Problem						

Fig. 4: Notification Requirements Table

Within the alarm group, we identified four 4 levels of severity by grouping common system responses described for each condition. We conceptualized the severity level category as parent modes and each condition with them as child modes. This allowed us to categorize the 18 individual conditions within one of the four levels and organize their requirements within the respective severity level. For example, all highest severity conditions caused the system to raise audio alarm, display error message and stop infusion, whereas all low severity level conditions caused the system to display an error message. Although we came across a couple of conditions whose requirements that did not exactly match the requirements of the four groups, we were able to negotiate with our domain experts to match the differences. To make the documentation clear, we used a table to document the conditions and the expected system responses, as shown in Figure 4.

This structuring greatly helped in systematically formalize each requirement. It helped provide enough information about the orthogonality and exclusivity among the modes. Infact, in some cases it helped identify and raise the right questions to discover missing or clarify ambiguous requirements. We were able to refer to the requirements document and formalize the requirements in a straightforward fashion, that otherwise would have been a ardours trial and error approach. For instance, by examining the

requirements hierarchically from the top level, we were able to systematically identify that the precise formalization of the patient infusion requirement includes the operating condition of the system (whether the system is ON/OFF) and its orthogonal alarm requirements to systematically identify exceptional conditions that avoids infusion and vice versa. This structuring significantly reduced the effort to arrive at the precise formalization. For example, the formalized requirements for patient bolus infusion was:

```
SystemOn and not(InfusionInhibitAlarms) and
   not(Configuring) and (PatientRequest) ⇒
        SoftwareFlowRate = PatientFr
```

Similarly, formalizing the alarm requirements to verify them individually also became straightforward. Grouping the exceptional conditions and its associated actions in the requirements document eased the process of formalizing the requirements. We used variables (or macros) to capture the conditions that cause that alarm level and the desired actions, to avoid repetition in definitions. It then became systematic and straightforward to individually check a requirement. For example, in the GPCA there were 3 exceptional conditions that we categorized as medium level alarms, namely air in line (sensor input indicating air in infusion tube), occlusion (sensor input indicating blockage in infusion tube) and door open (sensor input indicating drug reservoir enclosure is opened). To check the door open requirement exclusively, we systematically excluded the all higher priority level alarm (HiLevelAlarm is a logical disjunction of all conditions within the highest severity level) and the other conditions in its peer level, as shown below :

```
SystemOn and not(HiLevelAlarm) and
    not(AirInLine) and not(Occlusion) and
      (DoorOpen)⇒
         MedLevelAlarmActions
```

While such hierarchical structuring is common when trying to model the system or developing source code, it has not be widely used to document requirements in the requirements engineering community.

4.2 Solution For Challenge 2: Richer Traceability

When we identified this issue for one requirement, we started documenting the traceability between the system and software requirements. In addition to mapping each system requirement to the corresponding software requirement, we also documented an argument (as semiformal statements) explaining how the software requirement contributes to satisfying the system level requirement. While the original intent to document this richer traceability was to establish a satisfaction argument between requirements at different levels of abstraction, it actually helped us to understand the precise role of each component to satisfy the system requirement. In fact, by documenting this type

of richer traceability we were able to identify 10 incorrectly formalized software requirements out of the 86 system level requirements of the GPCA. In sum, documenting a richer traceability clarified the role of each component requirement in satisfying the system level requirement and the helps identify the mathematical difference between capturing them - one of the pitfalls in formal verification.

I want to put a figure from the traceability document for this requirement here.

4.3 Solution For Challenge 3: Sanity Checks and Fallibility Detection

We believe that the root cause of this problem was not the tool's default setting, but the fact that it was not apparent. Hence, to address the issue from the tool side we requested the tool developers to provide elaborate messages of certain key configurations when using the features. After the change, AGREE now displays the depth of check along with the result of consistency checking. While such a change may not be possible with every tool and feature we use, we strongly recommend the engineers to delve deep into the details and configuration of tool to precisely understand the results, especially when there is limited information provided by the tool along with the results.

4.4 Solution For Challenge 4: Automating Verifiers

I need to write a little about the tools above helps identify these issues + need translators and rigours process

5 conclusion

In this paper, we elaborated on the challenges encountered while adopting formal methods for engineering requirements of a complex medical device system.

I have to add more references

References