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. 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 formal methods has given rise to sophisticated techniques and tools, there is a lack of awareness and methodological guidance in using these techniques effectively, that often makes their use difficult and the results of their application leading to overconfidence into the correctness of the fielded system in its intended environment.
In this paper, we report on using formal methods to verify a complex infusion pump system. While the effort was very successful and has led to a complete verification of a hierarchically composed software architecture (down to the implementation of software components in Simulink), it was not without challenges that we believe are not adequately presented in the research literature. 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) some incorrect guidance exists on "flowing down" system requirements to lower levels of abstraction, (c) inexperience with tools can lead to "proofs" based on faulty premises, and (d) inadequate mitigation of risks when using multiple analysis tools can lead to misplaced confidence about the system. We then explain our approach to identify, mitigate and address such concerns.

1 Introduction

In safety critical systems, formal methods (mathematical techniques) can be used to rigorously assure the correctness of systems. However, 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

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

benefits of using formal methods [8, 12] as well as discussion about the limitations of formal methods [3, 4], there are not adequate discussions about the practical challenges encountered and techniques adopted to mitigate them in real applications. We believe that sharing such experiences will allow wider dissemination of formal methods in industrial use.

In previous work [9–11] we proposed scalable and efficient model based approaches to verify the formalized requirements of complex control systems, given their architecture (hierarchical decomposition into components) and formalized requirements. The approach involves hierarchically modeling, formalizing and verifying the requirements of the system and its components using multiple tools (Simulink, Simulink Design Verifier [5] and AADL/AGREE [1]), each with slightly different formal representations and usage. While this approach was undoubtedly useful to verify systems effectively, when we used it to verify an industry sized medical infusion pump system, 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 methods domains, have not been discussed in depth and addressed in any existing literature.

In this paper we describe such 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: Formalizing requirements for verification with respect to a system (or its model) not only involves translating the natural language statements into a mathematical notations but also requires one to systematically identify the contextual information for each requirements that, unfortunately, is not easily inferable from the traditional ways of documenting natural language requirements. In our effort to formalize the requirements of an infusion pump, that were were documented using standard patterns [6] and templates, we spent a number of hours to precisely identify the antecedent/precondition for each requirement. This was primarily due to the lack of guidance in systematically identifying the contextual information from the requirements documents. This triggered us to explore the root cause of this challenge and identify a structuring scheme to document requirements that would address this concern.

Requirements Refinement: Identifying component requirements from system requirements when the system is decomposed is not a simple breakdown of requirements but an analysis activity in which one ascertains if the component requirements are precisely allocated and if that allocation is sufficient to satisfy the system level requirements. While there are a number of tools to automatically and efficiently verify if such an allocation satisfies the overall system requirements, ensuring if the component requirements were accurate remains a challenge. While decomposing the infusion pump system, we initially mapped the system requirements to software requirements as is advocated in [7] (and used in other papers such as [**?**, **?**]), but found that applying this approach led to incorrect compositions when we tried to prove system requirements. These errors instigated us to define systematic practices and additional checks to detect and avoid these issues.

Tool Fallibilities: In an hierarchical 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. While formalizing the leaf level component requirements for the infusion pump, we unknowingly formalized some computations used within requirements in a inconsistent and unrealizable way. Unfortunately, the tools did not identify this error and successfully (and vacuously) proved the system level requirement. This prompted us to enhance the tool's capability towards checking *realizability* and *consistency* of requirements.

Matching Tool Boundaries While is a common approach to use a variety of tools and techniques to overcome the scalability concerns of formal verification, matching the tool boundaries in an informal or semiformal way often leads to elusive errors. While verifying the infusion pump system using different tools, we found that lack of formal mapping and automation at tool boundaries induced several errors that lead to misplaced confidence in results. Apart from errors related to semantic mismatches between the notations and tools, the mere action of transcription of requirements between different tools (even through the tools shared the same semantics) induced several errors that were not easy to detect. This motivated us to define formal sematic mapping, implement tools and identify rigours practices to systematically match the tool boundaries.

In sum, we assert that the benefits of formal methods in verifying requirements can be misleading, unless there is a change in both requirements engineering activities to support formal methods as well as in formal techniques to recognize requirement engineering concerns. In this paper, we report on the challenges and non-obvious nuances in using formal methods to rigours verify an infusion pump's requirements. We describe our approach to overcome those challenges from both requirements engineering and formal methods perspective. While we illustrate these lessons learned using the infusion pump we believe many engineers working on similar applications, especially in the safety critical system domain, will face similar challenges and hence we hope that sharing our experience proves 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 4, we elaborate our approaches and recommendations to address the challenges. Finally, we conclude the paper in Section 5.

2 Background

In this section, we describe our prior work [9, 11] 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 that allows patients to self-administer a controlled amount of additional drug. In order to analyse the problems associated with 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 safety features in the device to detect hazardous anomalies/behaviours and mitigate it 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, using model based techniques. We captured a reasonably complete set of overall system requirements using documentation patterns [6] and standard templates [**?**] that we thought will make the task of formalizing requirements straightforward in the verification phases. To cope with the complexity of modeling and verifying the GPCA we followed a hierarchical decomposition approach, that allows the modeling and verification to be systematically and rigorously partitioned into smaller and manageable tasks. In this approach, the system is hierarchically decomposed into a set of interconnected components (its architecture) and each component is allocated with its own set of requirements. At the leaf level, in addition to the allocated requirements, the detailed behaviour of each component is captured. This approach allows us to establish the requirements of the overall system by hierarchically verifying the component requirements at each level with respect to its model (the behavioral model at the leaf level and architecture model in all other levels).

We used a number of tools and techniques to implement this approach for the GPCA. We used Architectural Analysis & Design Language (AADL) to capture the architecture of the GPCA (hierarchical decomposition of components and its interconnections) and an extension of the AADL language that supports specification of formal textual requirements along with its respective components (and system) in the AADL model. While, the tools and notations in this approach allowed us to systematically allocate requirements to each component, it was not a suitable and easily understandable notation to model behaviours of the system. Hence, at the leaf level in addition to capturing its requirements using AGREE extension, we modeled the detailed behaviour of each component using MathWorks Simulink/Stateflow notation and tool. To verify the hierarchical architecture driven requirements decomposition, we used a compositional reasoning tool named AGREE [1]. Given the system architecture and the allocated requirements for both the system and its components, AGREE hierarchically verifies if the system level guarantees holds as a logical consequence of both its immediate child component requirements. AGREE uses a bottom up approach in which it verifies one level of decomposition at a time. Starting from the leaf-level it verifies if its parent level requirement are satisfiable given the leaf-level requirements. Using these verification results, it now automatically verifies the next higher level requirements and so on until the top most level is reached. To establish the leaf-level requirements in AGREE with respect to its detailed behavioural model modeled in Simulink, we used the Simulink Design Verifier tool. This involved recapturing the requirements in AGREE notation to a notation supported by Simulink Design Verifier. Fortunately, the recapture was straightforward since the notations used by both the tools were semantically same and syntactically similar. Hence, we manually recaptured the properties between the notations without any difficulty. This approach and tool chain allowed us to rigorously demonstrate an end-to-end system verification using tools and notations commonly used in the industry.

While our approach was very efficient and led to a complete verification of a hierarchically composed software architecture, the challenges in adopting the approach was not visible until we applied it ”at scale” involving multiple developers over an extended period of time. We initially demonstrated the above approach only using a small set of infusion pump safety requirements (18 requirements). Hence, to implement this approach to the entire GPCA, whose overall system complexity and size is comparable to a real industrial system, we allocated the task of formalizing and verifying them hierarchically in the GPCA model to a new team member, who had experience using these tools. We initially assumed this task was straightforward and planned 6 weeks for the activity. Unfortunately, we faced a number of challenges in the process of formally verifying the GPCA that we elaborate in the next section.

3 Requirements Engineering Challenges

In this section, we elaborate on four challenges that we believe is not limited to the GPCA case example but encountered by engineers working on similar applications, especially in the safety critical system domain. In our opinion, these challenges have the potential to undermine the rigor and usefulness of formal techniques yet has not be adequately discussed or addressed in any existing literature.

3.1 Challenge 1: Requirements Formalization

While model-based approaches helps verify requirements, the process of formalizing requirements from the natural language documents is often a error-prone and laborious process. Although, natural language (NL) has is the practical choice for capturing requirements when it comes to rigorously verifying them, formal notation is the solution. When the systems are complex - which is mostly the case - the process of formalizing requirements becomes a non-trivial activity. The challenge arises from both the ambiguity and implicit contextual knowledge in the NL statements. While the process of formalization implicity takes care of the ambiguity, precisely identifying he context of the requirement is a painful process. By contextual nature of requirements, we mean the specific state of the system in which the requirements need to hold; in formal terms it is the antecedent or precondition in a formal statement. Even the well known specification patterns focus on recapturing the NL statements to formal notations and does not help address the challenge of identifying the context of the requirements. To partially address this concern, in domains such as safety critical systems, the formalized requirements are verified with respect to a model of the system. When the context is not sufficiently captured in the formalized requirement, tools such as model checkers return counter examples to help engineers discover the contextual information. However, the counter example directed context exploration is a very time-consuming task, especially for certain types of requirements.

When formalizing the requirements of GPCA, we found that identifying the context for two groups of requirements was challenging and tiring. The first set of requirements were those that describe the behaviour of the system under under normal working conditions (those that were not safety requirements). 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 certain conditions in the system that prevented the patient bolus infusion to occur and hence the requirement was not satisfied in that context. These conditions were actually safety features of the system to prevent hazards that were documented as "alarm requirements" in another section in the requirements document. Similarly, consider another requirement that describes how basal infusion that is typically scheduled to infuse a certain quantity of drug over a specific period of time should operate. To verify this requirement, we needed to identify all the system conditions that can override the basal infusion. This included bolus infusions that have higher priority, hazards conditions that prevent infusion, clinician's manual pause or cancel that can temporarily suspend or abruptly stop infusion, etc. Again, these system conditions were present in the document in the form of other requirements, but organized in a manner that was unsystematic for formalization. Unfortunately, traceability between such requirements was neither available nor easily establishable and maintainable in practice, due to the numerousness of the requirements and the complexity in understanding the dependency between the behaviours they capture.

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

The second group of requirements that was troublesome were the mutually exclusive requirements 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 (a high priority) 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 (low priority), the system shall only display an appropriate message. One of the problems we encountered was trying to formalize it in such a way that their independent effect is verifiable. For instance, to verify a requirement with a lower priority condition we had to systematically capture the absence of all the higher priority ones. The challenge was systematically identifying the priority when it was not explicit specified. Unfortunately, there was no guidance or patterns to help us systematically organize and formalize such requirements for verification. On the contrary, the GPCA model 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 the undisciplined organization of the requirements statements within 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 formal tools help automatically verifying the sufficiency of component requirements to guarantee the system level requirement, deriving and formalizing those component requirements has been a manual activity. It is not impossible to derive and formalize a set of incorrect and/or infeasible component requirements that the formal tools will verify successfully [2].

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.

To address this concern for the software in the four variable model Miller et al extended the model as shown in Figure 1. 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

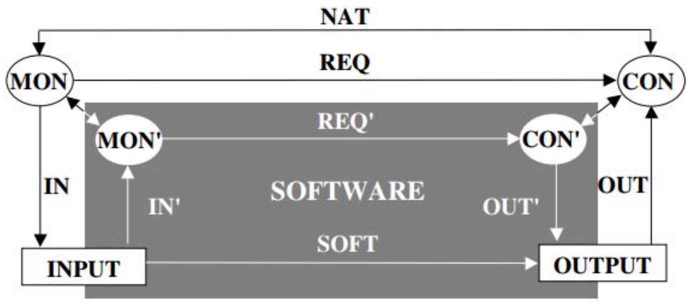


Fig. 1: Extension of Four Variable Model

function in REQ to REQ' using corresponding variables. They assert that this makes the tracing of the 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 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 the system requirement but in an unintended fashion. In reality, the software's output is deterministic and there was no need for the tolerance; that is the consequent in the formalization should have been `(FlowRate = BasalFr)`. That is, mathematically the requirement of the software is a function as opposed to the system level requirement that is a relation. In our opinion, failing to understand these differences in mathematical formulation leads to inaccurate understanding of formal verification.

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. This concern was not apparently visible until we started analysing how the software requirements realizes the system level requirements. It would have been beneficial if we had a tool that could have detected such anomalously verified 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 capability to check consistency of requirements in addition to verifying them that detects logical contradictions among requirements. The intent of performing such checks is to identify if the verification was trivially successful due to presence of inconsistent requirements. While the tool declared that all the GPCA requirements are consistent, our tool expert during the manually inspection phase found that there were several self-inconsistent requirements. To our surprise, those inconsistencies were not reported by the tool. This was primarily because we did not have full knowledge of the tool settings associated with that check.

Lets us illustrate the problem with an concrete example. To formalize certain requirements for GPCA we had to specify counters to 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. The formal language shown below is based on Property Specification Language (PSL) [16] and defines a Lustre language [11] flavor for the PSL expressions. Lustre is a synchronous dataflow language that describes the behavior of a system through a set of equations, and it can be viewed as a textual analogue to Simulink block diagrams. In this notation, it is possible to define constants, local variables and reusable fragments of temporal logic (called properties). Additionally, we can describe stateful relationships between variables using the 'pre' expression, which provides the value of a variable from the previous step of execution of the system. In the following formalization, "-¿" is used to capture the sequence of value computations for a variable. For example "0 -¿ 5" means the initial value is 0 and all the subsequent values are 5. We encoded the GPCA requirements using this notation.

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

While the tool successfully verified the above 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 20 such inconsistent formalizations in both system and component levels. To fix this issue, we used AGREE's operator *"pre"*, to refer to the value of a variable in the previous step as shown below.

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

While adding *"pre"* fixed this specific type of 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 (its initial value was 0), the consistency checker did not report the problem.

In our opinion, this situation is not limited to this specific example or tool but generalizable for all model checking tools and techniques. The root cause of this problem is not just our cursory knowledge of the tools, but also the lack of adequate details of the task displayed by the tool.

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,

$\texttt{SystemOn and (AlarmLevel = Med)} \Rightarrow \texttt{FlowRate} \leq \texttt{LowFlowRate}$

into AGREE, as follows:

$\texttt{SystemOn and (AlarmLevel = Med)} \Rightarrow \texttt{Flowrate} < \texttt{LowFlowRate}$

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.

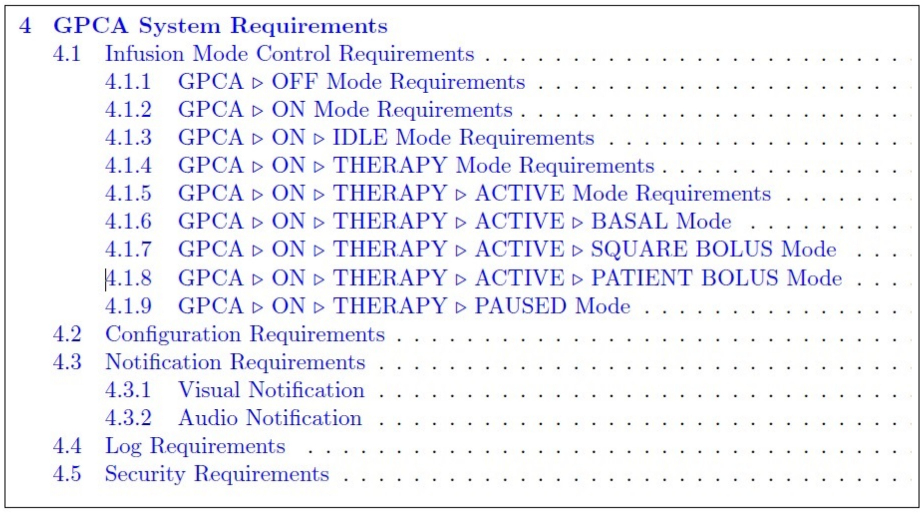
Another issue, that we believe is more common, is maintaining the synchrony between the requirements in both tools. After we proved requirements in AGREE, when we recaptured it for verification using SDV, at times we had to slightly change the way requirements were formalized, such that it is verifiable 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.



4 GPCA System Requirements

4.1 Infusion Mode Control Requirements

4.1.1 GPCA ▷ OFF Mode Requirements

4.1.2 GPCA ▷ ON Mode Requirements

4.1.3 GPCA ▷ ON ▷ IDLE Mode Requirements

4.1.4 GPCA ▷ ON ▷ THERAPY Mode Requirements

4.1.5 GPCA ▷ ON ▷ THERAPY ▷ ACTIVE Mode Requirements

4.1.6 GPCA ▷ ON ▷ THERAPY ▷ ACTIVE ▷ BASAL Mode

4.1.7 GPCA ▷ ON ▷ THERAPY ▷ ACTIVE ▷ SQUARE BOLUS Mode

4.1.8 GPCA ▷ ON ▷ THERAPY ▷ ACTIVE ▷ PATIENT BOLUS Mode

4.1.9 GPCA ▷ ON ▷ THERAPY ▷ PAUSED Mode

4.2 Configuration Requirements

4.3 Notification Requirements

4.3.1 Visual Notification

4.3.2 Audio Notification

4.4 Log Requirements

4.5 Security Requirements

Fig. 2: GPCA System Requirements Structuring

We restructured the requirement document and followed a hierarchical structure to organize requirements of the GPCA, as shown in Figure 2 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.

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

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. 3: Notification Requirements Table

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 3.

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

1. Darren D. Cofer, Andrew Gacek, Steven P. Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In Alwyn E. Goodloe and Suzette Person, editors, *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, volume 7226, pages 126–140, Berlin, Heidelberg, April 2012. Springer-Verlag.
2. Andrew Gacek, Andreas Katis, Michael W Whalen, John Backes, and Darren Cofer. Towards realizability checking of contracts using theories. In *NASA Formal Methods*, pages 173–187. Springer, 2015.
3. Anthony Hall. Seven myths of formal methods. *Software, IEEE*, 7(5):11–19, 1990.
4. Ralf Kneuper. Limits of formal methods. *Formal Aspects of Computing*, 9(4):379–394, 1997.
5. MathWorks Inc. Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier, 2015.
6. A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. Easy approach to requirements syntax (ears). In *17th IEEE Int'l Requirements Engineering Conf.*, pages 317–322. IEEE, 2009.
7. Steven P. Miller and Alan C. Tribble. Extending the four-variable model to bridge the system-software gap. In *Proceedings of the Twentith IEEE/AIAA Digital Avionics Systems Conf. (DASC'01)*, October 2001.
8. Steven P. Miller, Alan C. Tribble, and Mats Per Erik Heimdahl. Proving the shalls. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proceedigns of the Int'l Symposium of Formal Methods Europe (FME 2003)*, volume 2805 of *Lecture Notes in Computer Science*, pages 75–93, Pisa, Italy, September 2003. Springer.
9. Anitha Murugesan, Mats P.E. Heimdahl, Michael W. Whalen, Sanjai Rayadurgam, John Komp, Lian Duan, Baek-Gyu Kim, Oleg Sokolsky, and Insup Lee. From requirements to code: Model based development of a medical cyber physical system. *Fourth International Symposium on Software Engineering in Healthcare workshop (SEHC 2014)*, 2014.
10. Anitha Murugesan, Oleg Sokolsky, Sanjai Rayadurgam, Michael Whalen, Mats Heimdahl, and Insup Lee. Linking abstract analysis to concrete design: A hierarchical approach to verify medical CPS safety. *5th Int'l Conf. on Cyber-Physical Systems, 2014*.

11. Anitha Murugesan, Michael W. Whalen, Sanjai Rayadurgam, and Mats P.E. Heimdahl. Compositional verification of a medical device system. In *ACM Int'l Conf. on High Integrity Language Technology (HILT) 2013*. ACM, November 2013.
12. Michael W. Whalen, Darren D. Cofer, Steven P. Miller, Bruce H. Krogh, and Walter Storm. Integration of formal analysis into a model-based software development process. In Stefan Leue and Pedro Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2007.