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, 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) are often used to rigorously assure the correctness of systems. While advances in technology has improved the verification capability and efficiency of formal tools such as model checkers, they still rely on the skill level of the engineers to correctly formalize requirements and understand the technicalities of the tools. For instance, one could use a set of inconsistent (or incorrectly restrictive) premises to prove requirements using formal tools and gain misplaced confidence about the successful but meaningless verification. While there are numerous success stories that illustrate the benefits of using formal methods [11, 16] as well as discussion about the limitations of formal methods [6, 8], we believe that there are aspects of application of formal methods that remain necessary to

[⋆] This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

discuss. In our opinion, codifying the pitfalls will allow wider and better dissemination of formal methods in industrial use.

In previous work [12–14] 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 involved hierarchically modeling, formalizing and verifying the requirements of the system and its components using multiple tools, each with slightly different formal representations and usage. While this approach was undoubtedly useful to verify hierarchically decomposed systems, when we used it to verify an industry sized medical infusion pump system, we faced a number of practical challenges and pitfalls 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 capturing the natural language statements using mathematical notations but also identifying the contextual information for each requirements that, unfortunately, is not easily inferable from the traditionally documented requirements documents. In our effort to formalize the requirements of an infusion pump, we spent an enormous amount of time to precisely identify the context from its traditionally structured requirements document [1]. To overcome this challenge, we hierarchically organized the requirements that not only clarified the context for each requirement while formalizing them but also improved the clarity of the requirements document.

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 satisfies the system level requirements. While formal tools help verify the latter, verifying the former remains a challenge and lacks the right guidance. While decomposing the infusion pump system, we initially mapped the system requirements to software requirements as is advocated in [10] (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 motivated us to document a traceability argument that captures a description of how the component requirements satisfy the system level requirements that helped detect and avoid these issues.

Cursory Knowledge of Tools: In an architectural proof approach, all proofs are predicated on "leaf level" components meeting their specification. If these leaf-level specifications are incorrect (infeasible or inconsistent), 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, that was not detected by the tools and lead to misplaced confidence about the system. This prompted us to request for changes and enhancements to the tool to check *realizability* and *consistency* of requirements.

Matching Tool Boundaries In order to be able to reason at scale about complex systems, we found it necessary to use multiple reasoning tools at different levels of abstraction within the software architecture. This use necessitated translation of properties between multiple tools. 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. To have adequate confidence in the composition of the results, tools to translate between formalisms were required.

The contribution of the paper is two-fold: first, we report on the successful application of formal methods on a large and complex software architecture (the completion of the work in [14]). Second, we report on challenges that engineers will likely encounter while applying formal methods to complex control system requirements, especially in the safety-critical domain, and how we addressed these challenges. 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 **??**, we elaborate our approaches and recommendations to address the challenges. Finally, we conclude the paper in Section 4.

2 Background

In this section, we describe our prior work [12, 14] 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. Unfortunately, these devices have been involved in many incidents that triggered the need to strengthen their development practices [4]. In that context, our aim is to identify and demonstrate a rigours development approach of a generic infusion pump system using formal tools and techniques and release it publicly to 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, created a model of similar complexity to current generation commercial infusion pumps. 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 included advanced safety features in the device to detect hazardous anomalies/behaviours and notifications for clinicians and mitigations via inhibiting infusion and/or other device operations.

2.2 System Modeling and Verification

Our interest with the GPCA is to develop and demonstrate a rigorous end-to-end development approach, using model based techniques. To begin with, we captured a reasonably complete set of overall system requirements using documentation patterns [9] and standard templates [**?**] to simplify the task of formalizing requirements. During the modeling and verification phases, to cope with the complexity of the GPCA we followed a hierarchical decomposition approach that allows the modeling and verification to be systematically partitioned into smaller and more manageable tasks. In this approach, the system is hierarchically decomposed into a set of interconnected components (its architecture) and each component is allocated its own set of requirements. At the leaf level, in addition to the allocated requirements, the detailed behaviour of each component is modeled. 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 at that level.

We used a number of tools and techniques to implement this approach. We used the Architectural Analysis & Design Language (AADL) [15] to capture the architecture of the GPCA (components and its interconnections) and an extension of the AADL language to capture formal requirements of the components (and system) until the leaf level. While the AADL notations allowed us to systematically allocate requirements to each component, it is not designed for constructing component implementations. Hence, at the leaf level in addition to capturing its requirements, we separately modeled the behaviour of each component using MathWorks Simulink/Stateflow notation and tool [2] - a commonly used tool in the industry for behavioral modeling. To verify the hierarchical architecture driven requirements decomposition, we used a compositional reasoning tool named AGREE [3]. Given the system architecture and the allocated requirements, AGREE hierarchically verifies if the system level guarantees holds as a logical consequence of its component requirements. To establish the leaf-level requirements, we used Simulink Design Verifier tool [2] and verified them with respect to their detailed behavioural model. Since we used two different tools, we recaptured the requirements from AGREE notation to a notation supported by Simulink Design Verifier, namely Embedded Matlab [2]. Fortunately, the rewriting was straightforward since both the notations have the same underlying semantics (that of synchronous dataflow languages). This approach and tool chain allowed us to rigorously demonstrate a scalable end-to-end system verification.

While our approach led to a complete verification of a hierarchically composed software architecture, the challenges in adopting the approach was not apparent until we applied it ”at scale” over an extended period of time. In [14], we demonstrated a proof of concept analysis on an early version of the model over a subset of system requirements (18 out of the 86 system requirements). In both this effort and moreover, in the subsequent effort expanding the model and the set of functional system requirements, bringing on new team members, and validating our approach, we discovered a number of challenges that we elaborate in the next section.

3 Formal Requirements Analysis Challenges and Mitigations

In this section, we elaborate on four challenges that we believe are not limited to the GPCA case example but are typically 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 have not been adequately discussed or addressed in any existing literature. In the sequel we explain our approach in addressing them.

3.1 Challenge 1: Requirements Formalization

While model-based approaches, supported by formal reasoning tools, can be used to verify if implementation models meet requirements, the process of formalizing the requirements from the natural language documents is often a laborious process. Although, natural language (NL) is the practical choice for capturing requirements, formal methods are necessary to rigorously verify them. For complex systems, 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 the 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. While well known specification patterns focus on recapturing the NL statements to formal notations, they do not help to 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 counterexamples to help engineers discover the needed contextual information. However, this counterexample directed context exploration is a very time-consuming task, especially for certain types of requirements.

When formalizing the requirements of the GPCA, we found two groups of requirements whose context was particularly challenging to identify. The first set of requirements were those that describe the behaviour of the system under under normal working conditions. For example, one requirement states[1] that,

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

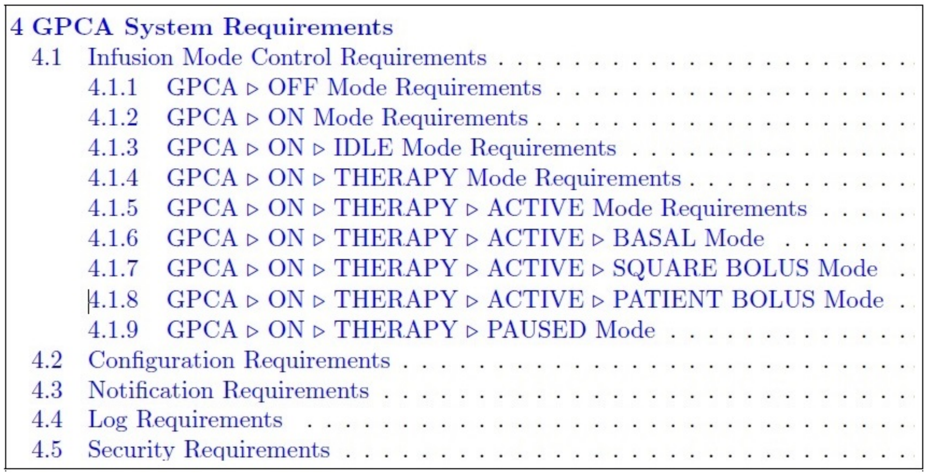
[1] We intensionally simplified this requirement such that it illustrates the problem. However, the original requirement had more conditions associated with it.

When we initially tried to formalize and verify the requirement, 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 from occurring 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. Unfortunately, traceability between such requirements was neither available nor establishing and maintaining then was practically straightforward, due to the numerous requirements and the orthogonality/exclusivity between the behaviours the requirements capture.

The second group of requirements that were difficult to formalize were those that were mutually exclusive, but had a certain inherent priority among them. In the GPCA, the alarm 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, the system responds to a high severity condition such as an empty drug reservoir by raising an audio alarm, displaying error message, and stopping infusion. On the other hand, when the system has been idle for a long time and a low severity level alarm is triggered, 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 problem was that there was no explicit priority 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 the undisciplined organization of the requirements statements within the document.

Approach: Structuring Requirements To systematically identify the context of each requirement and its dependencies with other requirements, we hierarchically restructured the GPCA requirement document as shown in Figure 1, that not only simplified the formalization process also helped understand the system in a conceptually clear way. The system behaviors of most control systems are typically captured by grouping them in terms of *modes* - a logical way to describe a set of system behaviours. In a complex system, there could be many modes that could either be mutually exclusive (only one mode can be active at a time) or orthogonal (multiple modes can be active at a time). By grouping the modes based on their common and exclusive behaviours as well as their interactions, we hierarchically organized them that helped systematically documenting, formalizing and verifying the requirements.

In the GPCA, there were three main orthogonal groups of modes - infusion, configuration and notification modes. Within each orthogonal group, there were a set of modes



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

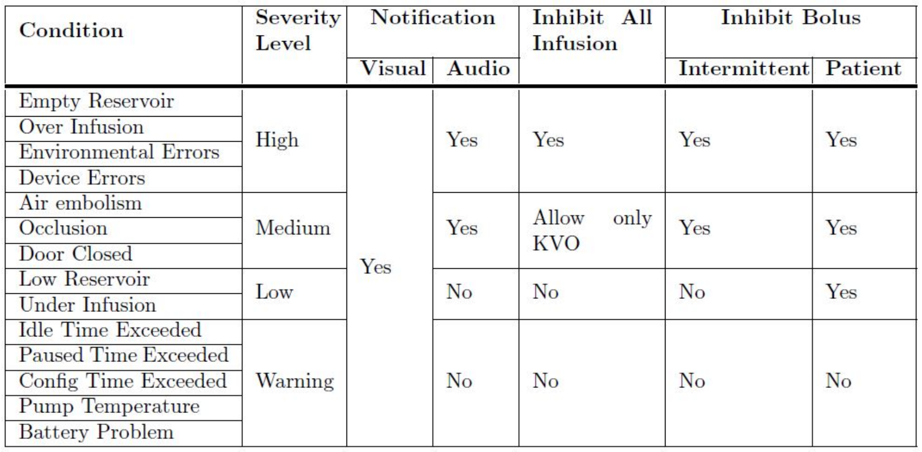
4.4 Log Requirements

4.5 Security Requirements

Fig. 1: GPCA System Requirements Structuring

that were mutually exclusive to each other such as basal, intermittent and patient bolus modes within infusion mode group. Similarly, the notification group had 18 requirements that we further categorized based on its severity level (Levels 4 to 1). When we analysed the requirements, we understood that there were many requirements that were common to the exclusive modes among an orthogonal group, as well as requirements that were common to all the three orthogonal mode groups. We leveraged this pattern of modal requirements to organize requirement statements within the document in an hierarchial fashion. In this structuring, at the top most level we placed requirements that were common to the entire system. At the next level, we placed the mode groups that were orthogonal to each other. Within each orthogonal mode group, we again followed the hierarchical structure to organize their requirements. For clarity purposes, within a mode group we grouped requirements hierarchically based on their behaviours and named it as a new mode (although it was not explicitly mentioned by the stakeholders). For example, we placed all the requirements that were common among the basal and boluses infusions in a parent mode called *Therapy* mode and made the basal and bolus its child modes with its own set of special requirements. For additional clarity on understanding the mutual exclusivity among modal behaviours, we documented requirements using a tabular notation. An example of the tabular format we used for the notification requirements is shown in Figure 2. Although we came across a couple of requirements that had minor variations to be grouped, this structuring helped identify those differences and we were able to clearly negotiate with our domain experts on appropriately categorizing them.

This structuring greatly helped in systematically determining the context of each requirement while formalizing it. It provided enough information about the orthogonality and exclusivity among the modes. For instance, by examining the requirements hierarchically from the top level, we were able to precisely identify and formalize the context of the patient bolus infusion requirement and the notification requirements that



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. 2: Notification Requirements Table

were discussed in Section 3.1. While such hierarchical structuring is common among modeling community, it has not been widely used to document requirements in the requirements engineering community. We believe that using such a strategy not only helps reducing the effort for formalizing requirements, but also provides the intellectual clarity to understand the system behaviours.

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 to automatically verify the sufficiency of component requirements to guarantee the system level requirement, deriving and formalizing those component requirements has been a manual activity. However, it is possible to derive and formalize a set of incorrect and/or infeasible component requirements that the formal tools will verify successfully [5].

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, Miller et al extended the four variable model, as shown in Figure 3. 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

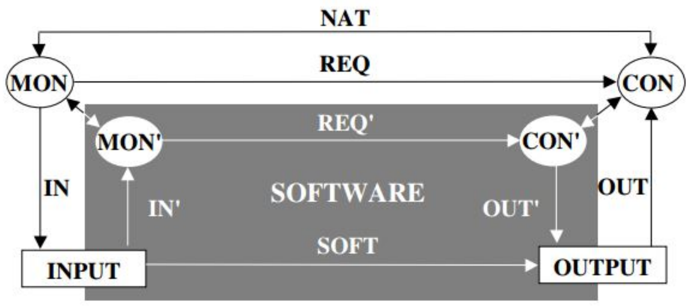


Fig. 3: Extension of Four Variable Model

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 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.

Solution : Traceability Argument To address this issue, we developed a *traceability argument* between requirements at adjacent levels in the hierarchy. Unlike the traditional traceability in which each parent level requirement is traced to its corresponding child requirement, we documented an argument of traceability that captures how child component requirements contribute to satisfying the system level requirement. This sort of assurance argument, that is inspired from Hammond's work on documenting justification for design decisions [7], helped us understand the role of each component in satisfying the system requirement. In fact, by documenting the *traceability argument* we were able to identify 8 incorrectly formalized component requirements of the GPCA. Further, this argument also helped us identify the component requirements that did not contribute to satisfying any of its parent level requirement. In sum, the traceability argument clarified the role and relevance of each requirement in satisfying its parent requirement, that in-turn helped identify the formalization errors in capturing them. While documenting this form of argument is not difficult when it is done at the time of requirements allocation, establishing this traceability after all requirements are allocated might be laborious. To address that concern, we are currently working towards building an automated traceability capability in the AGREE tool. This capability would automatically identify for every system requirement (parent level) the set of child component requirements that contributed to satisfying it. The task left for the developer is only to document the justification for the traceability.

I will see if I can include a picture of the traceability document here. I am just concerned about the space issues

3.3 Challenge 3: Cursory Knowledge of Tools

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.

Solution: Understanding Tool Settings We believe that the root cause of the AGREE tool not identifying the consistency issues in the formalized requirements was not the tool's default setting, but the fact that we were not aware of that setting and it was not apparent in the results. 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, that play a key role in determining their correctness. After that change, AGREE now displays the depth of consistency check along with the results. 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.

3.4 Challenge 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. In an earlier effort, in which we semi-formally mapped abstraction differences between infusion pump requirements formalized and captured as timed automata and AGREE notation, we identified semantic errors in the the mapping, in particular differences in the way time is handled in the notations. We fixed those issues by defining a formal semantics [17] to rigorously map the models and their requirements. However, we found that even when tools with "same" semantics were used rigorously maintaining the relationship between them in the long run is challenge.

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 comparable. 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 such issues - transcription errors and requirement management issues. The transcription errors occurred in the process of manually recapturing requirements. When recapturing some requirements we unknowingly changed the operators. In GPCA, we translated the following requirements that were proved in Simulink into AGREE as :

```
Properties in Embedded Matlab:
-----------------------------
sldv.prove(
    implies(SystemOn and AirInLine),(FlowRate <= MedFlow))
sldv.prove(
    implies(SystemOn and EmptyReservoir),(FlowRate <= LowFlow))

Property in AGREE:
----------------------------
guarantee "Property1":
    (SystemOn && AirInLine) => (FlowRate $=$ MedFlow)
guarantee "Property2":
    (SystemOn && EmptyReservoir) => (FlowRate $=$ LowFlow)
```

It was not possible to dismiss this error as a mere typo, since it demonstrates the potential for occurrence of such issues. Unfortunately, this error was not visible until we checked for feasibility using a recent enhancement of AGREE's tool called *realizability*. Intuitively, realizability checks if there is exists an output of the system for every input that satisfies the requirements. In the above case, the realizability check identified a counterexample in which both AirInLine and EmptyReservoir[2]. While this feature was not intended to identify the recapture errors, coincidentally we were able to identify this issue. After the found this errors, on manual inspection we identified 6 such transcription issues and corrected them.

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 sometimes missed to be changed and reverified in AGREE - another human error that caused mismatch between the requirements. 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, in our opinion, a lack of rigours process and/or automation to translate and check for synchrony is the root cause of this problem.

Solution: Automated Translators and Manual Inspections To address the transcription errors for GPCA, we first strengthened the process of verifying the translation using code inspection by a different developers. This inspection was effective in identifying errors, but was time consuming. In the long run, to maintain the synchrony between the requirements in both the tools was challenging. This challenge became the motivation for another team who are currently developing an automated translator that recaptures properties from AGREE tool notation to Embedded Matlab notation and vice-versa. While we are not actively involved in the development of the automation, we are currently verifying and validating their work using the GPCA's manual translation. While

[2] AirInLine indicates a hazard in which air bubble(s) are detected in the infusion tubing and EmptyReservoir is a hazard that indicates that the drug reservoir does not have sufficient drug to infuse

code inspections were effective, automated translators significantly reduced the overall time in the overall process.

4 Discussion and Conclusion

In this paper, we elaborated on the challenges encountered while adopting formal methods for engineering requirements of a complex medical device system. On analysing the root causes of each of the challenges, we assert that to realize the benefits of using formal methods to verify requirements we need adapt both the requirements engineering activities to support formal methods as well as the formal techniques to recognize common requirement verification concerns. In this section, we elaborate on how we addressed the first two challenges by changing requirements engineering process and the other two challenges by changing and enhancing the formal tools.

References

1. Ieee recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, Oct 1998.
2. The MathWorks Inc. http://www.mathworks.com, 2015.
3. Darren D. Cofer, Andrew Gacek, Steven P. Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In *Proceedings of the NASA Formal Methods Symposium*, volume 7226, pages 126–140, April 2012.
4. FDA. White Paper: Infusion Pump Improvement Initiative. April 2010.
5. Andrew Gacek, Andreas Katis, Michael W Whalen, John Backes, and Darren Cofer. Towards realizability checking of contracts using theories. In *Proceedings of the NASA Formal Methods Symposium*, pages 173–187. Springer, May 2015.
6. Anthony Hall. Seven myths of formal methods. *Software, IEEE*, 7(5):11–19, 1990.
7. J. Hammond, R. Rawlings, and A. Hall. Will it work? [requirements engineering]. In *Proceedings of the IEEE Int'l Symposium on Requirements Engineering*, pages 102 –109, 2001.
8. Ralf Kneuper. Limits of formal methods. *Formal Aspects of Computing*, 9(4):379–394, 1997.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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*.

14. 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.
15. SAE. http://www.aadl.info/aadl/downloads/papers/aadllanguagesummary.pdf.
16. 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.
17. Michael W Whalen, Sanjai Rayadurgam, Elaheh Ghassabani, Anitha Murugesan, Oleg Sokolsky, Mats PE Heimdahl, and Insup Lee. Hierarchical multi-formalism proofs of cyber-physical systems. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 90–95. IEEE, 2015.