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

Anitha Murugesan, Daniel 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 in 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 successful and has led to end-to-end 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 them 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 that could cause misinformed judgement about the correctness of the system, (c) inadequacy in understanding the technicalities of the formal tool can lead to "proofs" based on faulty premises, and (d) inadequate mitigation of risks when using multiple analysis tools can result in 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 inconsistently (or incorrectly restrictive) formalized 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 [19, 25] as well as discussions about the limitations of formal methods [9, 15], we believe that there are not adequate discussions about the practical challenges encountered and techniques adopted to mitigate them in real applications. In

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

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

In previous work [20–22], 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 useful to verify hierarchically decomposed systems, when we used it to verify an substantially complex 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 verification domains, have not been discussed in depth and addressed in any existing literature.

In this paper we discuss the challenges we faced while formally verifying requirements at multiple levels of hierarchy, applying tools and techniques "at scale" involving several developers over an extended period of time. We focus on four challenges:

Requirements Formalization: Identifying the contextual information when formalizing requirements from a traditionally structured requirements document is a challenging task. In our effort to formalize the requirements of an infusion pump, we spent an enormous amount of time to identify the context for each requirement, even though the requirements were organized as per standard templates [2]. This is mainly because the way of the requirement statements are linearly organized within the document did not help precisely identify its context. To overcome this challenge, we hierarchically organized the requirements within the document that not only clarified the context for each requirement while formalizing them but also improved the clarity of the document.

Requirements Refinement: When systems are decomposed and requirements are allocated to its components, establishing the fidelity between the system and its component requirements is a challenge, especially when system requirements are informally expressed and component requirements are formally specified. While decomposing the infusion pump system, we derived formal software requirements from the natural language system requirements using an approach proposed by Miller et.al [18] (also used in other papers such as [13,14,16]), but found that it led to incorrect compositions when we tried to establish system requirements. To address this concern, we propose documenting a *satisfaction argument*, originally described by Jackson [12], that captures the relationship between the system and component requirements as well the assumptions that are necessary to establish that relationship. Explicitly documenting this argument for every requirement allowed us to validate both the requirements allocated to the components and the related assumptions.

Cursory Knowledge of Tools: In an hierarchical architectural proof approach, all proofs are predicated on "leaf level" components meeting their requirements. If these leaf-level requirements 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 en-

hancements to the tool in checking the *realizability* [8] 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 reporting the challenges and non-obvious nuances in using formal methods to verify requirements of complex systems that, we believe, engineers will likely encounter. To some extent, although these limitations seem obvious once stated, in our experience, these issues are not fully realized and mitigated by some parts of the formal requirements analysis community. While we illustrate our 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.

2 Background

In this section, we provide a brief overview of an infusion pump system, its modeling and verification activities [20, 22] that is required to understand our work and the challenges described in the rest of the paper. Infusion pumps are medical devices that are used to deliver controlled quantities of the drug into patient's body. Unfortunately, these devices have been involved in many incidents that triggered the need to strengthen their development practices [7]. 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 analyze 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/behaviors 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 [17] and standard templates [2] to simplify the task of formalizing requirements. During the modeling and verification the GPCA, to cope with its complexity, 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 [22], the system is hierarchically decomposed into a set of interconnected components (its architecture) and each component is allocated its own set of requirements. We used the Architectural Analysis & Design Language (AADL) [24] to capture the architecture of the GPCA and an extension of the AADL language to capture formal requirements of the components (and system) until the leaf level. At the leaf level, in addition to the allocated requirements, the detailed behavior of each component is modeled. While the AADL notations allowed us to capture requirements of each component, it is not designed for constructing detailed behaviorial models. Hence for the leaf level components we modeled their behavior using MathWorks Simulink/Stateflow notation and tool [3] - an industry standard tools for behavioral modeling. To verify the hierarchical architecture driven requirements decomposition, we used a compositional reasoning tool named AGREE [5]. 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 verify the leaf-level requirements with respect to their detailed behavioral model we used Simulink Design Verifier tool [3]. Since we used two different tools at the leaf level, we recaptured the requirements from AGREE notation to a notation supported by Simulink Design Verifier, namely Embedded Matlab [3]. 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 scalably verifies hierarchically composed systems, the challenges in adopting the approach was not apparent until we applied it ”at scale” over an extended period of time. In [22], we demonstrated a proof of concept analysis on an early version of the GPCA model over a subset of system requirements (18 out of the 86 system requirements). In an effort to expand the application of our approach to the entire software of the GPCA, whose complexity and size was substantial, we formalized and verified all its requirements hierarchically. Among the 86 informally specified system requirements, we identified 55 requirements to be allocated (or flown down) to the software and formalized them (rest of the system requirements exclusively relied on hardware components). Since the software component was further decomposed into 8 sub-components to cope with complexity, we again flowed down requirements to its sub-components that resulted in 102 sub-component requirements. The artifacts developed in this effort is publicly available at [1]. In both the initial effort and the subsequent effort of expanding the scope of GPCA verification, 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, have the potential to undermine the rigor and usefulness of formal techniques yet have not been brought to light in any existing literature. In the sequel we explain our approach to 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. To partially address this concern, in domains such as safety critical systems, model-based approaches allow the formalized requirements to be verified with respect to a model of the system. When the context is not sufficiently specified, tools such as model checkers return counterexamples to help engineers discover the needed contextual information, as described by Miller et.al [19]. 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 behavior of the system 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* $patient\_flow\_rate$ *"*

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 practical to establish and maintain, due to the numerous requirements and the orthogonality/exclusivity between the behaviors they describe.

The second group of requirements that we found difficult to formalize were mutually exclusive requirement with an inherent priority among them. In the GPCA, the

[1] We intensionally simplified this requirement such that it illustrates the problem. The original requirement includes the tolerances for the flow rate.

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 (low severity level alarm) the system just displays an appropriate message. We found that formalizing these requirement in a way to verify their effect independently was challenging. 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 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). Formalizing and verifying each alarm condition requirement 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 behaviors. 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 behaviors 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 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 High to Warning). When we analyzed 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 additional clarity on understanding the mutual exclusivity among modal behaviors, we documented requirements using a tabular notation, as shown in Figure 2 for notification requirements. Although we came across a couple of requirements that had minor vari-

Fig. 1: GPCA System Requirements Structuring

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

ations in grouping, this structuring helped identify those differences and negotiate with our domain experts to clarify their category.

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 were discussed previously. 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 behaviors.

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. Unfortunately, it is has been possible to derive and formalize a set of incorrect and/or infeasible component requirements that the formal tools will verify successfully [8].

In an effort to provide guidance to engineers performing and formally verifying the system decomposition, Miller et al [18] propose that most system requirements can be recaptured into identical software requirements by slightly modifying the scope of the requirement. Their approach is based on the famous four variable model [23] 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 interrelationships required to reason about their correctness. 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 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.

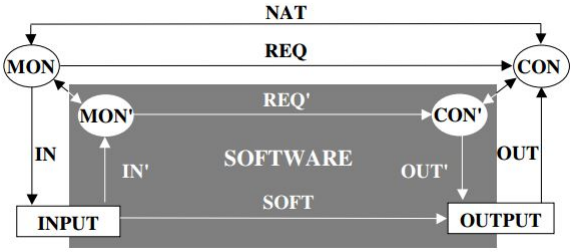

Fig. 3: Extension of Four Variable Model

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, that is they are inherently relational. For example, consider a system level requirement:

"In basal mode, the system shall infuse the drug at a flow rate within $basal\_flow\_rate$ $\pm t\%$ 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. When we wanted to identify software requirements of the GPCA, we followed the above guidance and formalized the software requirements in a way that 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 as per the guidance, the tolerance part was not intended to be allocated to the software. When we verified requirement it was successfully verified since the software indeed satisfies the requirement, 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 mere recapture presumes an implicit assumption that the physical components of the GPCA are perfect. This could lead to a situation in which if the software is compositionally verified in its environment (with physical components) it would fail. 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. While we were able to formally verify if the sub-components satisfies the software requirements, the correctness of the requirements allocated to the software was not formally establishable, since they were originally derived from informal system requirement. Unfortunately, the error in the requirements allocation described was not apparently visible until we started documenting an argument about how software requirements realizes the system level requirements.

Solution : Satisfaction Argument To identify issues with the requirements allocation we propose documenting *satisfaction argument* between each system (or component) requirement and its allocated component (or sub-component) requirements at adjacent levels in the hierarchy, that captures the relationship between them as well the assumptions that are necessary to establish that relationship. This sort of argument, that is inspired from Hammond's work [10], helped us understand the role of each component in satisfying the system requirement for GPCA. In fact, by documenting the argument we were able to validate the assumptions made for each requirement that consequently helped identify 8 incorrectly formalized component requirements of the GPCA software. While this documentation is straightforward when it is done at the time of requirements allocation, establishing the traceability between the requirements and its assumptions 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 analyze document the traceability and validate it. While we acknowledge that, if one could formally models and specifies requirements of every component of the system, we could use the tools to analyze the composition and identify the requirement errors. However, in practice, whenever informality is involved we recommend documenting a satisfaction argument for every requirement to validate the requirements flow down and assumptions.

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.

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

The formal language used to capture the requirements in AGREE is based on Property Specification Language (PSL) [11] and defines a Lustre language [4] "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 this formalization, "$->$" is used to capture the sequence of value computations for a variable. For example "$0-> 5$" means the value is 0 in the initial execution (or time) step and the value is 5 in all subsequent execution steps.

While the tool successfully verified the requirement involving the `boluscnt` variable, during manual inspection our tool expert pointed out this statement was internally inconsistent. The inconsistency was due to the usage of `boluscnt` in both the right and left side of the equation. According to the tool, the value for `boluscnt` always refers to its value in the current time step, whereas we actually intended to refer to the value in the previous time step in the right side of the expression. The incorrect formulation had caused a circular dependency in assigning values for `boluscnt` variable, that made this expression and the requirement involving this variable inconsistent and trivially satisfied. In the GPCA, we identified 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 being able to identify the consistency issues in the formalized requirements was because the tool had an incorrect 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 mapping, in particular differences in the way time is handled in the notations. We fixed those issues by defining a formal semantics [26] 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. For example, consider the following recapture of requirements between AGREE and Simulink:

```
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

[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

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. Further, performing inspections to maintain the synchrony between the requirements in both the tools in the long run was challenging. This challenge became the motivation for the tool developers to enhance their tool by 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 that automation, we are currently verifying and validating their work using the GPCA's manual translation. While code inspections were effective, we strongly recommend automated translators since they significantly reduced the overall time.

4 Discussion and Conclusion

In this paper, we elaborated the challenges that we encountered and our approaches to successfully use formal methods for engineering requirements of a complex medical device system. There were a number of useful lessons learned from this effort, especially when applying the techniques "at scale". Based on our experience, we assert that to realize the benefits of using formal methods to verify requirements we need to adapt both the requirements engineering activities to support formal methods as well as the formal techniques to recognize common requirement verification concerns.

The quality and clarity of requirements documentation plays a crucial role in cost effectively and precisely using formal methods to verify the requirements. While some of the existing requirements documentation patterns such as those proposed by Mavin et. al. [17] provide a set of structural rules to document requirements, they do not discuss about identifying the contextual information of the requirement. Even the well known *specification patterns* [6] focus on formalizing certain patterns of natural language requirements into temporal logic notations, but do not help address the challenge of identifying the context of the requirements. While we acknowledge the benefits of these approaches and advocate their usage, being able to precise context of the requirements plays a crucial role in formalizing and verifying the requirements. We believe that the hierarchical organization that we proposed in this paper is suitable for many

systems in the critical system domain. While the specific hierarchical pattern might vary between system, we suggest the requirements engineers to look for such patterns and appropriately structure their requirements.

In practice, mapping the informal requirements to formal statements, especially at different levels of system abstraction can not be completely avoided while developing systems. In such cases, establishing the fidelity between them is a challenge. Our recommendation to capture the *satisfaction argument* for every requirement, forces one to document how the component requirements and assumptions (if any) contributed to satisfying the system requirements. We believe that, this argument helps validate the correctness the requirements flown down to components as well as identify unrealistic assumptions with respect to every requirement.

While improving the quality of formalized requirements is the goal for the requirements engineers, we suggest that the developers of formal tools should identify *"hazardous"* use and fallibilities of their tools and mitigate them by either providing guidance or making them apparent in their tool's user interface. While we were able to request the developers of AGREE tool to enhance the capabilities of AGREE, we recommend the developers of formal tools to proactively build such capabilities.

In sum, we believe that the lessons we leaned in this effort was not limited to the infusion pump, but are broadly applicable to practitioners working on related applications and that our experience serves informative.

References

1. Generic patient controlled analgesia infusion pump project - http://crisys.cs.umn.edu/gpca.shtml.
2. IEEE recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, Oct 1998.
3. The MathWorks Inc. http://www.mathworks.com, 2015.
4. P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM Symp. Principles Program. Lang. (POPL)*, pages 178 – 188, Munich, Germany, 1987.
5. 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.
6. Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE, 1999.
7. FDA. White Paper: Infusion Pump Improvement Initiative. April 2010.
8. 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.
9. Anthony Hall. Seven myths of formal methods. *Software, IEEE*, 7(5):11–19, 1990.
10. 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.
11. IEEE. *IEEE Std. 1850-2005. Property Specification Language (PSL)*. IEEE, 2005.
12. Michael Jackson. The world and the machine. In *17th Int'l Conf. on Software Engineering*, pages 283–283. IEEE, 1995.

13. Ralph D Jeffords, Constance L Heitmeyer, Myla M Archer, and Elizabeth I Leonard. Model-based construction and verification of critical systems using composition and partial refinement. *Formal Methods in System Design*, 37(2-3):265–294, 2010.
14. Marjo Kauppinen, Juha Savolainen, and Tomi Mannisto. RE theory meets software practice: Lessons from the software development trenches. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 265–268. IEEE, 2007.
15. Ralf Kneuper. Limits of formal methods. *Formal Aspects of Computing*, 9(4):379–394, 1997.
16. David L Lempia and Steven P Miller. Requirements engineering management handbook. *National Technical Information Service (NTIS)*, 2009.
17. 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.
18. 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.
19. 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.
20. 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.
21. 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*.
22. 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.
23. David L. Parnas and Jan Madey. Functional documentation for computer systems engineering (volume 2). Technical Report CRL 237, McMaster University, Hamilton, Ontario, September 1991.
24. SAE. http://www.aadl.info/aadl/downloads/papers/aadllanguagesummary.pdf.
25. 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.
26. 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.