A Reference Model For Hierarchical Requirements

A DISSERTATION

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Anitha Murugesan

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Doctor of Philosophy

Dr. Mats P.E. Heimdahl, Advisor

Dr.Sanjay Rayadurgam, Co-Advisor

January, 2020

Acknowledgements

There are many people who have contributed to the successful completion of this dissertation. I would first like to express my sincere gratitude to my advisor Prof. Mats Heimdahl for his continuous feedback, support, patience, and motivation throughout the course of my Ph.D study. Without his assistance and dedicated involvement in every step throughout the process, this dissertation would have never been accomplished. I could not have imagined having a better advisor and mentor.

Besides my advisor, I would also like to thank Dr.Sanjai Rayadurgam for all the stimulating feedback, insightful suggestions and professional guidance. I owe a lot to him for being instrumental in helping me perfect the formal aspects of this dissertation.

I am also indebted to Dr.Mike Whalen for all his time, help, and the wisdom he has shared with me over the years. I also want to thank Dr.Neha Rungta and Dr.Oksana Tkachuck for giving me the opportunity to intern at NASA.

I would like to thank all my friends in CriSys research group for their feedback, cooperation and friendship. I cherish the awesome time that I spent with them in discussing research, daily life, and our future. Last but not the least, I would like to thank my family for unconditionally supporting me throughout this arduous but fruitful journey.

Dedication

I dedicate this dissertation to the four pillars of my life— my parents, my husband, my kids and God— for the unconditional love, patience, support, trust and blessings they endowed on me.

Abstract

Reference models such as Parnas' Four Variable model, Jackson's World-Machine model, and Gunther et al.'s WRSPM model abstractly define and relate key artifacts in requirements engineering. In the past decades, when systems were typically defined and developed in the traditional top-down way, these models were enormously useful for engineers to rigorously define and analyze their requirements. However, nowadays these models don't seem to be as helpful in discussing requirements of systems.

Modern systems are developed in a *hierarchical, middle-out and distributed* manner; i.e. starting from candidate architectures, they are iteratively composed using a set of existing or newly and independently developed subsystems. This has necessitated the need for engineers to define requirements at progressively varying levels of scope within the system, hierarchically co-evolve those requirements with the architecture, and apportion them to parts of the system such that each part is independently governable, yet composable. Unfortunately, hierarchically scoping, refining, and reasoning about requirements and architecture in an intertwined manner is both a conceptually and methodologically challenging, error-prone task. While the use of reference models has been a time-honoured approach to systematically address such challenges, none of the existing models discuss the notion of hierarchy or co-evolution between requirements and architecture. Nevertheless, capturing this hierarchical, co-evolutionary relationship in a generic framework such as a reference model, we believe, will be practically helpful for engineers.

In this dissertation, we define the *hierarchical requirements reference model* that abstractly, yet formally, explains the "Twin Peaks" of requirements and architecture. The goal of this model is to provide an unambiguous framework—a set of vocabulary and

rules—that engineers can use as a frame of reference to rigorously discuss these artifacts in real system developments. Central to this model is the notion of *hierarchy* that weaves together the requirements and architecture in a way that naturally allows understanding their attributes and relationships as well as addressing the related challenges.

Further, we defined a new formal definition of traceability based on the hierarchical satisfaction relationship. This way of formally defining traceability allowed us to formulate the notion of "complete traceability"—the ability to identify all possible satisfaction trace links between requirements. Further, it helped bring out the distinction between "necessary" vs. "sufficient" sub-systems' requirements to achieve a system-level requirement. This new insight into traceability, we believe, will greatly help engineers manage requirements of large complex systems.

To illustrate the practicality of this model, we use an infusion pump case example system —an industrial case study from the medical domain. Using a model-based approach we describe how the concepts defined in this new reference model help rigorously define and reason about the device's hierarchic requirements and architecture. As we illustrate, we also describe our novel enhancements/extensions to existing requirements techniques and tools to allow hierarchical requirements capture and analysis. While the specifics of this illustration, such as the use of formal methods-based tools and techniques are particular to the case example considered, the reference model concepts are applicable to both formal and informal requirements engineering of modern, complex systems.

Contents

List of Figures

Chapter 1

Introduction

The hardest single part of building a software system is deciding precisely what to build. - Fred Brooks [8]

Complex systems are naturally constructed in a *hierarchical*, *heterogeneous* and *middle-out* fashion; i.e., they are designed and developed by iteratively composing newly developed and/or existing components of varying sizes and capacity starting from candidate architectures [4, 35]. For example, to design and construct a modern aircraft such a Boeing 747, thousands of its physical parts, millions of lines of its software and several computing resources have to be necessarily reused, purchased, and/or developed and assembled as a federation of sub-systems [5, 10].

A critical challenge in such developments is precisely engineering the *requirements*—descriptions of "what" needs to be achieved. While the requirements orthodoxy states that requirements of a system should be defined prior to and independently of its construction, it is generally impossible for engineers to do this in practice. As described in the *Twin Peaks* model [70], in middle-out developments, engineers typically *co-evolve* the requirements along with its *architecture*—the structural aspects of system. For example, the architectural decision to include sensors in an aircraft is influenced by the

requirement to avoid obstacles, whereas the choice of a specific make/model of the sensor may constrain and alter the overall requirements on categories of obstacles detected and the reliability of detection. While architecture was not traditionally considered as a requirements driver, in current practice however, engineers have to iteratively negotiate the requirements and architecture in an *intertwined* manner to bring them in accord.

Further, as systems are iteratively decomposed into smaller manageable components, their overall system requirements are flown-down and allocated to the components such that the overall system requirements are collectively met [35]. Consequently, requirements of such systems are not a single and independent artifact as traditionally conceptualized, but rather a set of artifacts that are hierarchically defined and refined per the system's decomposition (or architecture) [16]. In other words, the requirements and architecture of such systems are necessarily hierarchic and related to each other.

This close-knit relationship induces several challenges and issues that engineers have to deal with conceptual clarity in practice, such as:

- ***Determining the System Boundary***: It is conventional wisdom to identify (at-least conceive) a boundary between the system to-be-developed and its environment, so that one can precisely define requirements based on that boundary. However, in hierarchical middle-out developments where components of varying sizes and capabilities pre-exist at varying level of system detail, carving out such a boundary that may unevenly cut (conceptually) across multiple-levels of system detail and specifying requirements based on that is, nothing but, confusing. For instance, a publicly available infusion pump requirements document that contained requirements ranging from user needs to hardware and software specifications was ambiguous and confusing for stakeholders to understand, review, analyze, design or develop [66]. Hence, developments where multiple distinct teams develop parts of a system, there has to be a conceptually clean way to carve out the boundaries

and specify the requirements/architecture.

- ***Understanding "Your What is My How"***: As engineers recursively decompose the system, descriptions of each component that is considered as design ("How") at a certain level becomes the requirements that are refined and flown-down to components ("What") at a lower level [89]. Hence, at every level of decomposition, meticulous book-keeping of the level of detail and its flow-down becomes crucial [16]; otherwise, details can easily get mixed-up and cause confusion.

- ***Performing Verification & Validation***: Assuring the consistency, completeness and correctness of requirements is challenging, yet essential in many systems, such as those in the safety critical area [95]. However, to do so in hierarchically constructed systems, engineers have to systematically and seamlessly compose "what" the sub-systems do (their requirements) based on their composition (the architecture). Further, in systems developed middle-out, the composition includes existing and newly developed components. With an increase in the system's size and complexity, this becomes a challenging and error-prone task for engineers to perform. Unfortunately, there have been several examples in the safety critical domain where imprecision in such analysis have lead to misplaced confidence in the safety of systems [56, 55].

- ***Capturing Requirements Traceability***: A common approach taken by engineers to keep track of system's requirements flow-down across multiple levels of system detail is recording (often manually) a cross-reference between them. This is called *requirements traceability*. It helps understand how a high-level requirement is realized by the lower-level components—and conversely, how a requirement allocated to a lower-level component help satisfy a high-level requirement.

> With architecture and requirements being hierarchically intertwined in complex systems, traceability is more than merely relating requirements; rather, one has to relate it with the architecture [15] in an hierarchical, seamless manner. Lack of such precise traces have been known to severely impact change analysis and maintenance of systems later in their life-cycle [36, 23].

Unfortunately, there is not adequate support available for engineers in the requirements engineering literature to help rigorously understand and deal with the challenges of the close-knit relationship between requirements and architecture in practice [57]. Nevertheless, addressing these challenges, we believe, is crucial to ensure that requirements-related issues are identified early in system development; after all, the root cause of most catastrophic accidents caused by safety-critical systems have been attributed to poor requirements [55, 52]. In that respect, we **hypothesized** that clarifying the requirements-architecture relationship of systems constructed the hierarchic, middle-out way in a fundamental, yet rigorous manner will provide a consistent means to understand and deal with these challenges in practice.

This dissertation is an attempt to rigorously explain the hierarchical intertwined relationship between requirements and architecture in-terms of a *reference model*. In general, reference models are abstract frameworks that capture core concepts and essential relationships in an application domain, while neglecting specifics, so that they can be used as reference across a large class of systems in that domain. In the 90s researchers such as Parnas [74] and Jackson [45, 31] defined such reference models. During the time when most systems were developed the traditional way, these models largely served as a frame of reference for engineers and helped them address several requirements related concerns that existed then. However, the models abstracted away the notion of hierarchy in architecture and its intertwined-relation with requirements. In fact, the models were fundamentally based on principles of traditional development where requirements

are defined and perfected before the architecture is discussed. So, the models do not lend themselves naturally well to address the above mentioned challenges with modern system. Hence, we defined a new requirements reference model that accounts for the architecture.

Central to our reference model— called the *Hierarchical Reference Model*— is the notion of hierarchy among the requirements of a system that is established per the system's architectural decomposition. This model explicitly captures the intertwined relationship between requirements and architecture. In particular, this model focuses on the *hierarchical satisfaction* relationship that is established between the requirements of a system (or sub-system) and the requirements that gets allocated to its sub-systems (or sub-sub-systems). In this framework, we formally define fundamental properties of the requirements such as consistency (not contradictory), realizability (possible to implement) and acceptability (satisfy sufficiently) by systematically weaving-in the architecture as well as rigorously accounting for the notion of time. This formulation allows capturing two novel properties *sustainable consistency* that establishes preservation of consistency over time and *decomposition realizability* that captures the logical coupling between the requirements and the architecture.

By conceptually organizing the requirements along the architectural decomposition of the system, the model highlights the idea that system decomposition is both an architectural and requirements exercise. By capturing hierarchy, the model supports the notion of changing roles of artifacts between levels of abstraction. Precise guidance on when and how to determine the relevance and influence of an artifact while hierarchically engineering systems is an essential piece of guidance that this model provides. By hierarchically relating requirements and architecture, the model allow seamless end-to-end analysis of the system requirements. Additionally, such analysis helps determine how a particular requirement is indeed satisfied by its sub-components and assess whether the

sub-components were allocated more requirements than needed or over-constrained.

While the hierarchical reference model theoretically describes the core concepts, we acknowledge that engineering challenges come into play when practically applying it to real systems. To concretely explore such challenges we chose a non-trivial case study from the safety critical domain— an infusion pump, a medical device system. Using the reference model as a guide, we define requirements and decomposition for the infusion pump using state-of-the-art model-based techniques.

Further, we have defined a new formal definition of traceability based on the hierarchical satisfaction relationship. This way of formally defining traceability allowed us to formulate the notion of "complete traceability" – the ability to identify all possible satisfaction trace links between requirements. Further, it helped bring out the distinction between "necessary" vs. "sufficient" sub-systems' requirements to achieve a system-level requirement. This new insight into traceability, we believe, will greatly help engineers manage requirements of large complex systems.

1.1 Objective and Summary of Contributions

Our **long-range goal** is to improve the rigor of requirements engineering (RE) in the development of complex systems, particularly in the safety critical domain where numerous mishaps and accidents have occurred due to requirements-related issues. Though requirements of complex systems are hierarchically intertwined with the system's architecture in practice, requirements are dealt with separately from the architecture of systems. This is reflected in most of the existing approaches and all reference models. Hence, the **objective** of this dissertation is to address both the theoretical gaps and practical challenges in accounting for system's architecture in RE. We have defined a new reference model that clarifies the hierarchical intertwined relationship between requirements and architecture in a formal manner. Based on the model, we have outlined

rigorous techniques to specify, verify, trace and assess requirements of complex systems.

The **intellectual merit** of the presented research lies in the new conceptual framework that formally captures the mutual influence between requirements and architecture. While the synergy between requirements and architecture has been a well-discussed topic in the past decade, to the best of our knowledge, little/no work has been done in terms of rigorously aligning them in theory and practice. The results of our research is a significant step towards potential improvements to the RE for complex systems, particularly those in the safety critical domain. Requirements engineers in this domain would use our model as a frame of reference to understand and define requirements of real-world systems. Further, they could also use the techniques and tools we defined either as-is or as a guide to engineer the requirements. Such adoption could result in significantly fewer number of challenges in dealing with complex system requirements and fewer requirements-related errors.

We have made the following specific contributions to the requirements engineering literature in this dissertation:

Defined a new reference model: We have defined a *"Hierarchical Reference Model"* that provides an abstract framework to discuss the requirements of a system and its intertwining with architecture. The artifacts, their essential attributes and relationships are formally defined in this model to unambiguously understand the concepts and to reason about the artifacts. The goal of this model is to provide an abstract framework that engineers can use as a reference to rigorously define and analyze requirements of actual systems. The way we have defined the key concepts in this reference model has allowed us to capture properties that help identify issues that are unique to requirements-architecture interplay.

Formulated novel approaches to hierarchically specify and verify requirements: Backed by the strong theoretical concepts defined in the model and using novel

combinations and extensions of existing techniques and tools, we devised RE approaches for large, complex systems with hierarchical architectures. In particular, to cope with the practical challenges posed when engineering requirements of such systems in the safety critical domain, we defined scalable approaches using model-based techniques to systematically identify, specify and verify the requirements and architecture in a hierarchically layered manner.

Established a unique, rigorous notion of requirements traceability: Based on the *hierarchical satisfaction* relationship among requirements that we defined in the model, we established a formal explanation for requirements traceability (relationship between requirements). This allowed us to explore the distinction between establishing a single trace between requirements and all trace links and define a notion of *complete traceability*. In addition to formally defining this idea as a proof of concept, we have also extended an existing formal verification tool to precisely extract the trace links.

Illustrated the ideas using a case study: We use an industrial strength case study from the safety critical domain— an infusion pump system—to illustrate all of the above. Through this illustration we show how the new model and the approaches naturally helps deal with the challenges of engineering modern system requirements. The artifacts of this case study are made publicly available to serve as exemplars for the RE community.

1.2 Structure of This Document

The remainder of this dissertation is organized as follows. In **Chapter 2** we outline the background of requirements in practice and survey closely related requirements reference models. In **Chapter 3** we present the our new hierarchical reference model that formally capture the requirements-architecture relationship. In **Chapter 4** we

introduce the infusion pump system, a medical device that we use as a running case example throughout the dissertation. In **Chapter 5** we discuss various model-based approaches that help systematically exploring, specifying, reasoning and organizing hierarchical requirements of complex systems using the infusion pump as a case example. In **Chapter 6** we explain our layered and scalable approach to formally verify the hierarchical requirements and illustrate it using the infusion pump. As an extended application of the reference model, in **Chapter 7** we present a novel definition of satisfaction based traceability (using concepts defined in the reference model) and discuss its practical ramifications of this way of formally establishing trace links between requirements. **Chapter 8** concludes the dissertation and discusses potential future work.

Chapter 2

Background and Related Work

The heart of every scientific discipline is its own unique, uniform and acknowledged terminology. – Oliver Thomas

The word *Requirements* is one of the most frequently used terms in system development. Yet, different people in this field seem to have different interpretations of what requirements are and are not. Consequently, it raises several questions such as *What, why, when and how are the requirements established and related in complex systems?* In this section, we explore these questions in practice and discuss the literature that have attempted to rigorously explain them.

2.1 Requirements in Practice

Colloquially, requirements are defined as someone's necessity or want. In system development, such needs are typically elicited from a set of people or organizations who pay for a system to be developed to meet their needs. For instance, consider a patient monitoring system that *shall notify the nurses-in-charge as soon as the patient's heart beats less than a certain number of beats per minute.* Such statements that describe the actual needs to be achieved are called *User*, *Customer* or *Business Requirements.* It is

typically captured using terms and concepts that are well understood by the people in the domain such as the heart beat and notifying the nurse.

Unfortunately, the domain/people-centric terms used to capture requirements could lead to ambiguity and unintelligibility for those who would build a system to achieve those requirements. For instance, a system can not directly deal with heart beat, rather it can only detect physically measurable quantities such as heart sound, its vibration, or pulse. Instead of letting the developers decide, analysts/engineers identify such system-centric quantities and recapture the user requirements in terms of those quantities. Such recaptured statements are called *system requirement specifications* or just *specifications* and they are passed to the system development teams. Along the same lines, in complex software-based systems, the requirements for the software portion of the system are also separately documented as *software requirement specification* in addition to user and system requirements.

Though all of this seem straightforward when explained using simple examples, as the complexity of a system increases the set of terms and requirements grow in sheer numbers and complexity. The distinction/relationship between them becomes challenging to understand and manage. In fact, several systems have failed due to poorly specified requirements and specifications as well as inappropriately used terms [34]. To address these problems, researchers in the 1990s (and earlier) [74, 45, 31] defined fundamental concepts of requirements mathematically, so that engineers can unambiguously use them as a reference when dealing with requirements.

2.1.1 World-Machine Model

Jackson's World-Machine (WM) model [45] is one of the foundational reference models in RE and establishes the distinction between requirements and specification. The model defines a conceptual separation between the *Machine*— the system to be developed,

the *World*— the environment in which the system is going to be installed, and the *Interface*— the interaction space between the World-Machine (Figure 2.1). Based on this separation, the World-Machine model defines the following artifacts:

- **Requirements (R)** – descriptions of the user's needs in a certain environment expressed in-terms of concepts/terms existing in that environment;
- **Domain knowledge or Assumptions (W)** – the presumed facts and assumptions about that environment.
- **Specification (S)** – description of the system to be developed expressed in-terms of concepts/terms shared between the system and the environment.

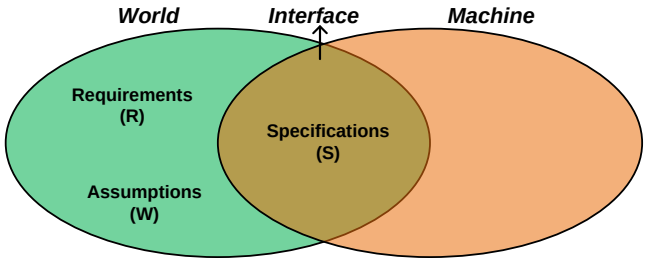


Figure 2.1: World Machine Model

Further, the model defines a relationship between these artifacts, called the *satisfaction relation*, that establishes the adequacy of the specification (S) to the meet (denoted by $\vdash$) the requirements (R), given the environmental assumptions (W),

$$S, W \vdash R$$

The abstract, yet rigorous, definitions of requirements and its relationships provided the clarity for researchers and practitioners to understand and discuss requirements of complex systems, as well as further investigate the interconnections between problem definition and solution exploration [85].

WRSPM Model

Gunter et al. [31] expanded the WM model and augmented its reasoning rules to allow rigorous formal reasoning of requirements. The model expands the WM model with two additional artifacts, P(program) and M (its platform), as illustrated in Figure 2.2; so, there are five artifacts (W-R-S-P-M) in this model.

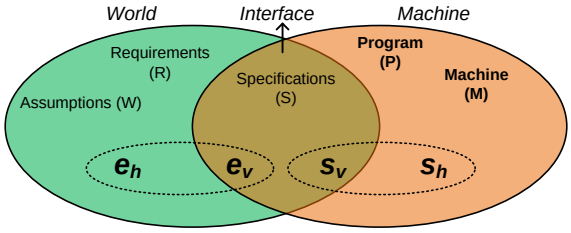


Figure 2.2: WRSPM Model - Extended World Machine Model

This model defines the notion of *phenomena*—states, events, or individuals— that belong to the World and the Machine, and categorizes them in terms of their visibility and control. The phenomena (say e) that belong to the World is further classified into those that are hidden from Machine (e_h) and visible to the Machine (e_v). Similarly, the phenomena that belongs to the Machine (s) is further classified into those that are hidden from the World (s_h) and visible to the World (s_v). Based on this classification, the model defines each artifact as a mathematical relation expressed over a subset of the phenomena: W and R are expressed over e and s_v, S is expressed over e_v and s_v and P and M are expressed over e_v and s.

The main contribution of this model lies in the way the essential relationships between the artifacts were formalized using the distinction between the phenomena used to express them [34]. Namely, *domain adequacy* establishes non-trivial existence of environment in which requirements should be satisfied; *adequacy* establishes satisfaction of requirements by the composition of specification and assumptions; and *Relative Consistency* helps ensure the consistency between the specification and its environment.

Let us illustrate some practical benefits of these models, by instantiating the model in-terms of the example, as shown in Figure 2.3[1].

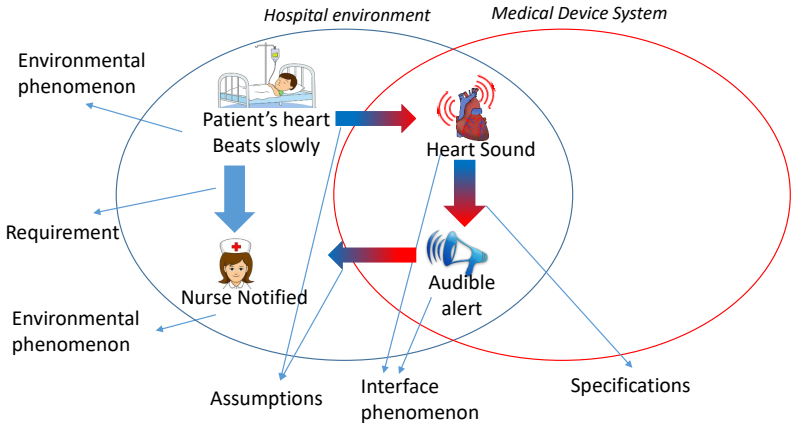


Figure 2.3: Requirements-Specifications-Assumptions

Fundamentally, grounding the terms and using them to define the requirements, specifications, and assumptions helps to remove ambiguity and resolve scoping issues among the statements. Further, the satisfaction argument helps engineers understand the importance of capturing assumptions; without properly capturing needed assumptions the satisfaction argument cannot be completed.

2.2 Coping with Complexity

To facilitate the comprehension of complex systems and allow development practices such as distributed development, reusability, integration with existing systems, and plug in commercially developed components [75, 70], it is a common practice to decompose systems into smaller and manageable parts or *components*. For example, the medical device can be decomposed into sensor(s) that read environmental inputs, controller(s) that commands control actions based on the inputs, and actuator(s) that realizes the

[1]The ellipses in the figures are used to represent the scope of the world and machine as per [45].

control actions in the environment, as shown in Figure 2.4.

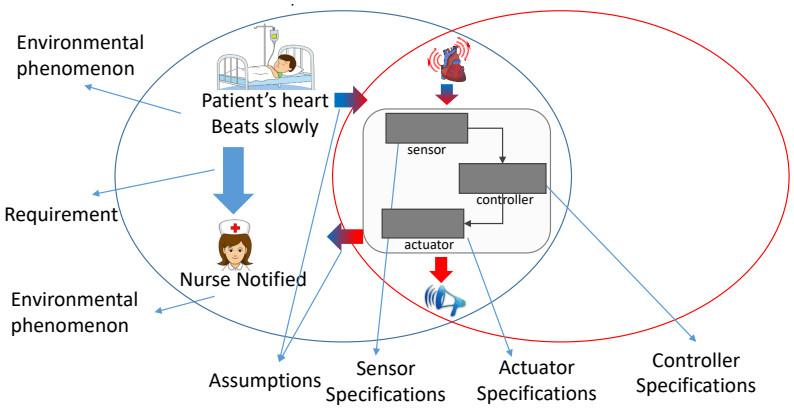


Figure 2.4: Decomposition

To enable (independent) development of each of these components, their descriptions are also captured separately to distinguish their roles and responsibilities. For instance, one can often find *software* and *hardware specification* artifacts that exclusively describe the needs of the respective components. When defining these component-level specifications, engineers can be faced with conceptual confusions and errors in understanding, maintaining a clear separation between the component-level requirements-specifications-assumptions and analyzing their properties, similar to the issues encountered at the system-level. Recognizing this as a enduring conundrum in most control-systems development, Parnas defined a model to serve as an abstract framework to discuss and reason about artifacts of multi-component systems [74].

2.2.1 Four Variable Model

The Four Variable Model by Parnas et al. [74] is an influential model that, in fact, predates the World-Machine and WRSMP models. The Four Variable Model provides mathematical guidance for defining control system requirements. The striking feature of this model is the practical perspective it brings by generalizing the representation of

a system in-terms of the following five relations (REQ, NAT, IN, OUT, and SOFT) and four sets of variables (monitored–m, controlled–c, input–i, and output–o), as shown in Figure 2.5:

1. **Requirements (REQ)** is expressed over the monitored (m) and controlled (c) environment variables,
2. **Nature (NAT)** are constraints imposed by the environment and other assumptions, expressed over m and c,
3. **Input (IN)** is representing the sensors of the system, defined as a relation over monitored (m) and the sensed input variables (i),
4. **Software (SOF)** is the relation describing what the software (controller) in the system must do; it is defined as a relation over the sensed input (i) and output variables (o),
5. **Output (OUT)** is representing the actuators of the system, defined as a relation over output (o) and controlled variables (c).

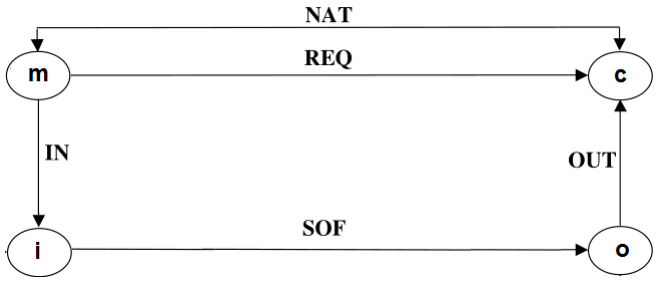


Figure 2.5: Functional Documentation Model

In late 1990s and early 2000s this model was widely used in many software based safety critical control system developments since it helped conceptualize the system and easily matched the high-level design of a majority of systems. Further, the mathematical notations to express the concepts allowed rigorous formal reasoning of the requirements.

In fact, this allowed the automation of the reasoning process that resulted in significant cost savings and improved quality of the system [41, 42].

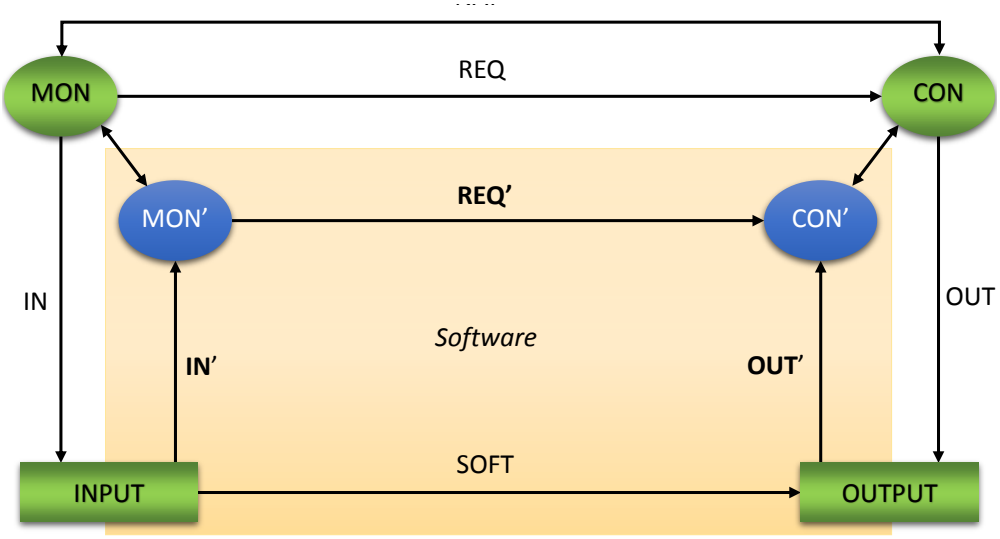


Figure 2.6: Extended Four Variable Model

As an extension to this model, Miller et al. [62, 82] proposed the "Extended Four Variable model", as shown in Figure 2.6. In this work, the authors point out that the weakness of the four-variable model is that the software requirements ($SOFT$) is defined by specifying NAT, REQ, IN and OUT. In complex systems, drivers or wrappers abstract away the mapping details between the actual hardware and control software, hence they extend the reference model with relations IN' and OUT' (the specification of drivers). They also recognize that there are always some differences between the monitored and controlled variable values in the environment and the value that is sensed and set as input and outputs. To accommodate the differences, they recreate internal (to the software) versions of the variables (m' and c') and restate REQ in terms of these virtual variables, REQ'. The main contribution of this model is the distinction between actual sensors, actuators and their wrappers that are defined in

practice, that had been a common source of confusion [40]. It also makes the tracing of the *REQ* to the software direct and straightforward.

2.2.2 The Hierarchy

While simple on-level decomposition may suffice for developing certain systems, large and complex systems such as aircraft, cars, and complex medical systems, are typically developed in *hierarchies*—a structure that allows the system to be discussed in progressively increasing levels of detail. Hierarchy results from iterative decomposition of components into sub-components (2.7), typically to cope with complexity and accommodate middle-out-development. This decomposition induces the need to hierarchically define the requirements of the components and its sub-components.

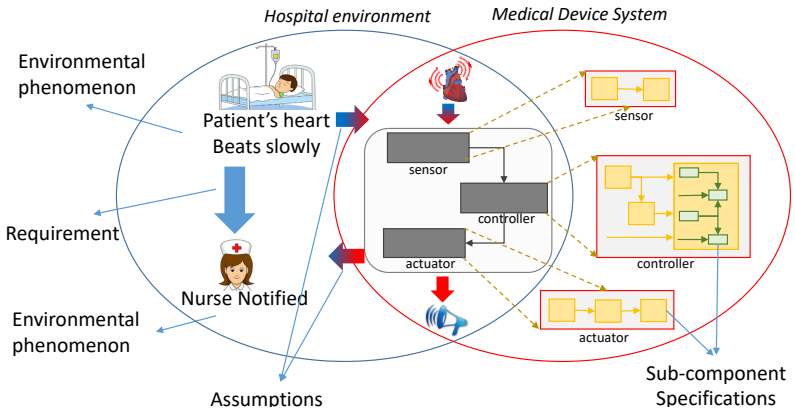


Figure 2.7: Hierarchy

Analogous to hierarchical flow-down of requirements, KAOS—a Goal Oriented Requirements Engineering Framework [84], discusses hierarchical refinement of *goals*—prescriptive statements of intent about a system-to-be developed. The central idea of KAOS is the notion of refinement-abstraction hierarchies where high-level user goals are iteratively refined into sub-goals until each sub-goal is realizable (or assigned to)

an *agent*—an entity that is responsible for satisfying that sub-goal, for example, software, hardware, or even humans and the environment. KAOS has a graphical tool to visualize the hierachy as well as a underlying formal notation that helps automatically analyzing the goal refinement. While KAOS speaks of hierarchical decomposition and has a formal approach to reason about it, there is no discussion regarding the strategy or rationale that guides the decomposition.

Only a few researchers and practitioners have recognized the hierarchical flow-down of requirements with respect to the system architectural decomposition. One prominent example is the REVEAL Process [35] that provides an informal practical requirements approach to deal with hierarchy in large complex systems. This method explains from a practical perspective how large systems are decomposed into smaller manageable components in which each component is allocated with its set of requirements in such a way that the composed system satisfies its overall requirements. They highlight the notion that this allocation is not a mere partition of system requirements but an analysis effort that has a justification associated with it. At every level of decomposition, the system requirements are related to its components' requirements with this justification. This relation, termed *Rich Traceability*, provides an informal explanation to the system decomposition and requirements flow down for reasoning about the system. The REVEAL approach is influenced by Jackson's World-Machine Model.

While the discussion of the REVEAL Process brings to light the need for a justification when decomposing systems and allocating requirements, it does not discuss in depth the relationship between the requirements and the decomposition itself. If we examine the act of decomposing a system into components (and then assembling the components into a system), it induces a requirements analysis effort in which one needs to ascertain whether the requirements allocated to sub-components are sufficient to establish the system-level requirements. Equally important, one also needs to determine

whether any limitations of a component or its assumptions need to be accounted for at the system level to make sure the overall system performs as expected. This is shown informally in Figure 2.8.

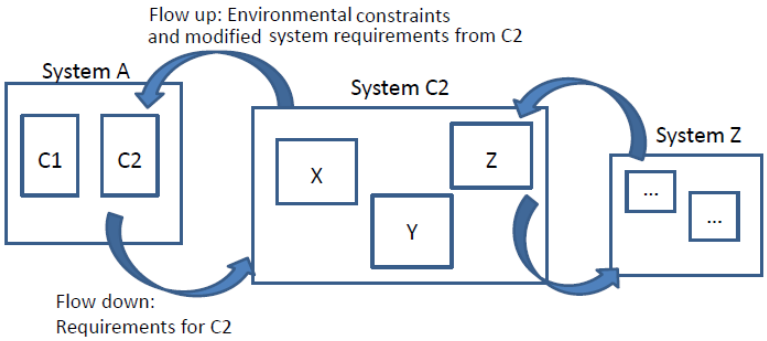


Figure 2.8: Interplay

The flow-up can either translate to new and altered requirements of the system, or could be allocated to the environment as assumptions on how the system will be used. As we begin to allocate requirements to components, we may find that the architecture we have chosen simply cannot meet the system-level requirements. This may cause us to re-design the system to allow us to meet the system-level requirement, levy additional constraints on the external environment, or to renegotiate the system-level requirement. Hence, the decomposition of a system touches both requirements and design concerns. In essence, it is evident that the system design is an essential aspect to consider while engineering the requirements of a system. Yet, in typical developments, the process of requirements and decomposition are approached as if they are two independent activities and their relationship is not meticulously captured. While the REVEAL process focuses on documenting the rationale for the decomposition, it does not discuss how one could rigorously define and reason about the requirements flown down in the hierarchy.

A second well-known discussion on requirements refinement and hierarchy can be found in Leveson's Intent Specification [54]—an approach intended to help engineers

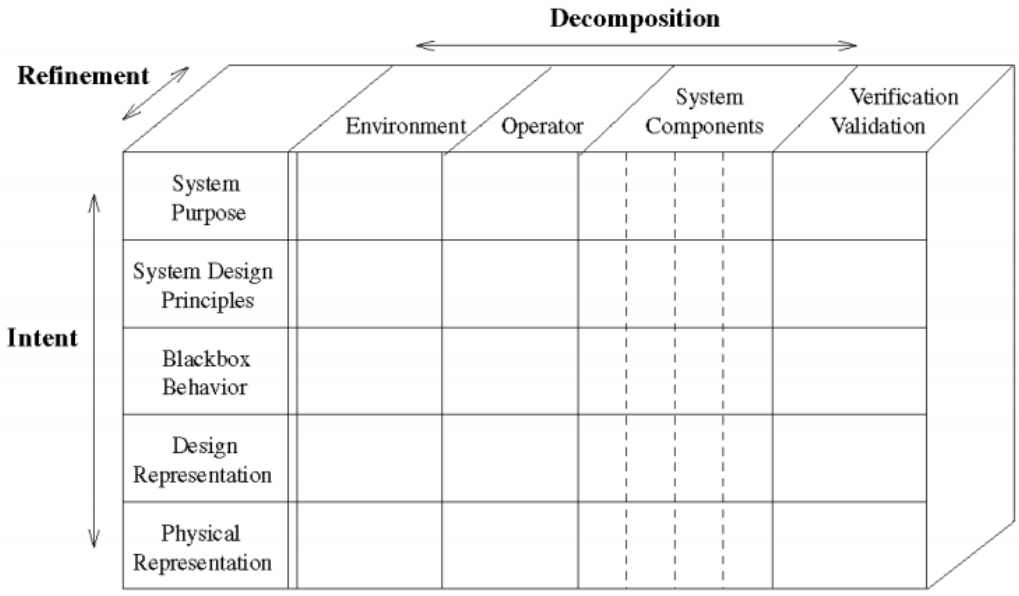


Figure 2.9: Intent Specification

specify and design systems hierarchically based on notions from both system engineering and psychology. The approach is to organize system-engineering artifacts, such as requirements and design, over three dimensions, intent, decomposition, and refinement. The vertical or *Intent* dimension consists of six levels, each of which provides rationale (or "why") about the level below. Along the horizontal or *Decomposition* dimension, the system in decomposed into heterogeneous parts such as Environment, Operator, System and Components. The third dimension, *Refinement*, further breaks down both the Intent and Decomposition dimensions into details. The information at each level is organized in such a way to allow reasoning and traceability across the hierarchical levels. A commercial document oriented tool, SpecTRM (Specification Toolkit and Requirements Methodology) [52], uses this methodology as its foundation. While this approach explains that requirements and design are decomposed and refined, it does not explain the cohesive relationship between them.

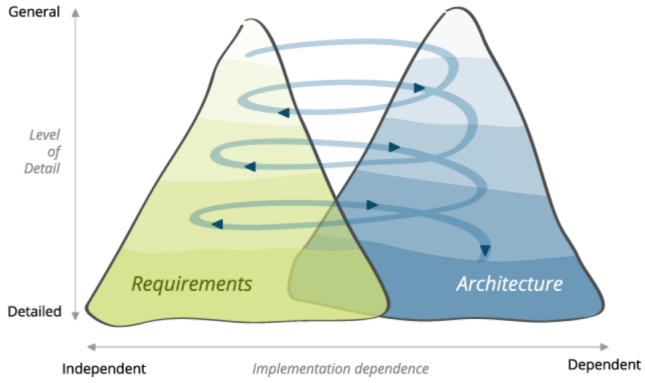


Figure 2.10: The Twin Peaks

In fact, it was only the Twin Peaks model by Nuseibeh [7] that elaborately, though informally, discussed the crucial relationship between requirements and the decomposition of a system. The model asserts that most system development starts from candidate architectures that have been used in similar systems. Further, to cope with cost, schedule and other constraints, components are often reused and/or procured. Hence, although the overall user requirements drive the high-level architectural decisions, the choice of specific architectural components restrict the set of achievable requirements. By comparing the process of co-evolution to spirally traversing two mountains (Figure 2.10), this model explains how it is healthy—and often a necessity—to create a sound architecture and correct requirements.

Building on the Twin Peaks model, we propose that the relationship between the requirements and the architecture is hierarchically intertwined since the architecture of complex systems is often hierarchical. Considering the extent to which RE has been explored until now, one might expect that there will be some sort of guidance or approach to help specify and analyze requirements and architectural artifacts in-tandem in an intellectually clean and rigours manner. Surprisingly, “the requirements” on a system are discussed as separate from, and more abstract than, “the architecture”

of that system both in practice as well as in the research literature.

2.3 Discussion

Undoubtedly, all the reference models and related work have set a strong foundation to discuss complex system requirements. Nevertheless, they do not comprehensively and completely help dealing with the modern middle-out, hierarchical developments.

The Four Variable model serves as a useful guidance to rigorously engineer requirements of traditional control systems. But it does not seem to lend itself well for modern systems with hierarchical architectures. Such architectures induce the need to meticulously capture the interconnection between components, specify requirements at multiple levels in the system hierarchy and seamlessly maintain their relationship. Unfortunately, there is no direct support in the Four Variable model to account or analyze these complexities. Though engineers could use this model to understand the way control system works at a high-level, we have found that it is not adequate as a guidance to rigorously engineer requirements of such systems in practice.

Along the same lines, the World-Machine model serves as a conceptual reference to fundamentally understand and distinguish between the requirements, specification and assumptions of a system under-development. An invaluable contribution of the WRSPM model is providing formal descriptions of the artifacts and their relationships. Namely, they define a property called *relative consistency* that helps formally reason about the consistency of the systems specifications with respect to the environment where it will be installed.

But, neither reference model discusses the complexities of hierarchical construction nor provides a comprehensive treatment of the crucial notion of time. Nevertheless, both of these play a crucial role in precisely engineering requirements of modern systems. At

a conceptual level, these models advocate defining a boundary between the system-to-be developed and its environment. In practice, however, such a boundary would be a conceptually convoluted one; this is because in hierarchical, middle-out construction, components pre-exist at multiple levels of abstraction and, hence, only a few components needs to be developed and its requirements specified. Thus, the conceptually clean notion of an environment and a machine is often not present in practice.

Although not explicitly mentioned, these models are fundamentally based on a traditional development paradigm where requirements are well-established (defined and perfected) prior to designing the system, and such requirement serve as the basis to define all other development artifacts. However, in practice, the initial form of requirements captured from users is like a "rough sketch" that gets honed only as one starts delving into the details of the system and exploring the environment. The way entities are structured in the environment and the way the components form a system architecture influences the requirements and that does not naturally let the requirements, assumptions, and specification be captured as a monolithic artifacts. The fact that the notion of (de)composition (or architecture) is completely abstracted away from the requirements definition, makes these model difficult to use as a guide for requirements engineering.

In summary, the related work discussed so far have made irrefutable contributions for the specific problems they were aimed to address. They do, however, not address the following:

- The interplay between requirements and architecture that occurs during progressive abstraction and refinement of the system.
- The need to flexibly represent the concrete, actual system architecture without imposing a rigid decomposition pattern.

- The definition of the essential relationship between requirements and architecture to allow seamless and rigorous analysis of the entire system.

A reader might naturally raise the question, *"Can't the models be extended as-is to accommodate hierarchy?"*. Our answer is "No". There has to be several fundamental changes to the way requirements and architecture have been conceptualized in these models. In the next chapter, we introduce a new reference model that will satisfy the above needs while retaining the benefits and elegance of the existing models.

Chapter 3

Hierarchical Requirements Reference Model

A formal representation should be as simple as possible, but no simpler – Pamela Zave and Micheal Jackson [93]

In this chapter we define our new reference model. Central to this model is the notion of *hierarchy* among the requirements of a system and its relationship to the system's architecture. When systems are designed as a hierarchical composition of components, the requirements are inevitably decomposed and flown-down to each component such that at each level in the hierarchy the composition of the component requirements *satisfy* the enclosing component's requirements. The goal of this model is to provide an abstract framework that engineers can use as a reference to understand and analyze this intertwined satisfaction relationship between the requirements and the architecture of systems.

3.1 Preliminaries

We begin by defining some fundamental terms that we will use throughout this chapter to explain concepts of the reference model.

3.1.1 Quantities and Types

Any observable and quantifiable properties, states or attributes of entities (or individuals) of interest in the problem domain, e.g., the heart beat of humans, the operational mode of vehicles, the temperature of a place, an input to a software component, etc., are called *quantities*. Each of these quantities is designated with a name and associated with a specific *type* that denotes the possible values it can hold. For example, we may choose integer for the heart beat, an enumeration for the operational modes, and Boolean for the software input.

Our interest is with quantities whose values are recorded as functions of time, that is, there is a value for the quantity appropriate for its type at every time instant. We consider time to be a quantity of type $\mathbb{R}^+$. If we denote the type of a quantity x as Υ_x, then at each instant of time $t \in \mathbb{R}^+$, the quantity's value that we denote by $x(t)$ is an element of its type, i.e., $x(t) \in \Upsilon_x$. Hence, we formally define a quantity as a function from $\mathbb{R}^+$ to its type,

$$x : \mathbb{R}^+ \to \Upsilon_x \tag{3.1}$$

3.1.2 Runs

While quantities may have values at all time instants, the set of their values over a desired interval or instances in time are called *runs* (as discussed by Westman [88]).

Mathematically, we define a run of a quantity x over time $T \subseteq \mathbb{R}^+$, denoted by θ_x^T as,

$$\theta_x^T = \{(t, x(t))\}_{t \in T} \tag{3.2}$$

The run of a vector of quantities, $V = (x_1, x_2, ..., x_n)$, is defined as,

$$\theta_V^T = \{(t, x_1(t), x_2(t), ..., x_n(t))\}_{t \in T} \tag{3.3}$$

In the rest of this chapter, to avoid tedium, we may skip the subscript T, superscript V, or both, when those are clear from the context and denote the run as either θ, θ_V or θ^T instead of θ_V^T; it is implicit that θ has an associated set of quantities (V) and time (T). In such cases, we will use the functions $V = \mathcal{Q}(\theta)$ and $T = \mathcal{T}(\theta)$ to discuss the quantities and times of that run respectively.

Using concepts from relational algebra, we now define certain operations that can be performed on runs and some relationships between runs. These operations and relations help explain the concepts of the reference model later in this chapter.

Join

The operation of merging two runs whose elements match in time but contain different (distinct) quantities is called *join*. The join operation results in a run whose elements are tuples having the same time instants as the original runs and the cross-product of values of quantities from the original runs. This operation is analogous to natural join operation over relations in relational algebra. Intuitively, a join operation allows discussing simultaneous valuation of the quantities of two (or more) runs over the same instances in time. Mathematically, the join (denoted by $\bowtie$) of two runs θ_V with $\theta_{V'}$

whose $\mathcal{T}(\theta_V) = \mathcal{T}(\theta_{V'}) = T$ is defined as,

$$\theta_V \bowtie \theta_{V'} = \{(t, x_1(t), x_2(t), ..., x'_1(t), x'_2(t), ...)\}_{t \in T} \tag{3.4}$$

where $V = \{x_1, x_2, ...\}$, $V' = \{x'_1, x'_2, ...\}$ and $V \cap V' = \emptyset$.

Projection

The opposite of join, called *Projection*, is an operation that removes a subset of quantities from each tuple of the run. The result of projection is a run whose elements are tuples that have the same time instants as the original run and the values of a subset of quantities. In formal terms, given a run θ with $T = \mathcal{T}(\theta)$ and $V = \mathcal{Q}(\theta)$, its projection (denoted by $\theta|_{V'}$) to a subset of quantities $V' \subseteq V$, is defined as an operation that removes the valuation of quantities in $V \backslash V'$ from each tuple in θ,

$$\mathcal{Q}(\theta|_{V'}) = V' \tag{3.5}$$

where $V' \subseteq V = \{x_1, x_2, ...\}$.

Extension

While the join operation concerns expanding the set of quantities of a run, *extension* deals with extending the run to more time instants (without altering the number of quantities). In essence, the notion of extension allows discussing super-sets of a run. Hence, we define *extension* as a relation between between two runs, θ_V^T and $\theta_{V'}^{T'}$ (to be read as θ_V^T extends to $\theta_{V'}^{T'}$):

$$\theta_V^T \subseteq \theta_{V'}^{T'} \Leftrightarrow (V = V') \wedge (T \subseteq T') \tag{3.6}$$

When the run is extended to only future time instances, it is called *future extension*

is denoted by $\trianglelefteq$. In formal terms,

$$\theta_V^T \trianglelefteq \theta_{v'}^{T'} \Leftrightarrow (V = V') \wedge (T \subseteq T') \wedge (\forall\ \tau' \in (T' \backslash T) \cdot \forall\ \tau \in T\ \cdot\ \tau' > \tau) \tag{3.7}$$

In natural language, when every time instant considered for extension (indicated by $(\forall\ \tau' \in (T' \backslash T)$ the above equation) is a future point in time than every time point of that run ($\forall\ \tau \in T\ \cdot\ \tau' > \tau)$), then the extension is called future extension.

3.2 The Reference Model

Having established the formal definitions of quantities, runs and their operations, we now turn to the definition of the hierarchical reference model.

3.2.1 Components and Quantities

The basic building blocks of this model are called *components*. Any independently conceivable and behaviorally cohesive entity related to a system of interest (to-be/already developed)—such as the system, its components, sub-components and its environment—is regarded as a component. This approach to uniformly regard all entities as components, irrespective of their proportion or constitution, allows the model to be relatable to real system development at any level of detail or specifics.

Every component interacts with its surrounding (wherever present or installed) through quantities of interest that exist in that surrounding. In particular, the component observes or senses certain quantities and actuates or produces others. Let us call them *monitored* and *controlled* respectively (borrowing the names from the Four Variable model [74]).

As noted by authors of other reference models, explicitly identifying, designating

and distinguishing the quantities (into monitored and controlled) for components helps delineate their scope and precisely describe their behaviors [31].

Since we will be referring to multiple components at a time in the rest of this chapter, we will use a distinct naming scheme to refer to a component and its quantities. Each component is assigned a unique name that is sub-scripted to its quantity names. For instance, if we denote a component by name β, its monitored and controlled quantities will be denoted by $m_{1_\beta}, m_{2_\beta} \ldots m_{i_\beta}$ and $c_{1_\beta}, c_{2_\beta} \ldots c_{j_\beta}$, where i and j are the number of monitored and controlled quantities of β. We also consider each quantity to be unique and distinct from others to avoid ambiguities [74], i.e., $m_\beta \cap c_\beta = \emptyset$ where $m_\beta = \{m_{1_\beta}, m_{2_\beta} \ldots m_{i_\beta}\}$ and $c_\beta = \{c_{1_\beta}, c_{2_\beta} \ldots c_{j_\beta}\}$.

Using the general definition of quantities in equation 3.1, the monitored and controlled quantities of a component β are defined as:

$$m_{1_\beta} : \mathbb{R}^+ \rightarrow \Upsilon_{m_{1_\beta}} \qquad m_{2_\beta} : \mathbb{R}^+ \rightarrow \Upsilon_{m_{2_\beta}} \quad \ldots \quad m_{i_\beta} : \mathbb{R}^+ \rightarrow \Upsilon_{m_{i_\beta}} \tag{3.8}$$

$$c_{1_\beta} : \mathbb{R}^+ \rightarrow \Upsilon_{c_{1_\beta}} \qquad c_{2_\beta} : \mathbb{R}^+ \rightarrow \Upsilon_{c_{2_\beta}} \quad \ldots \quad c_{j_\beta} : \mathbb{R}^+ \rightarrow \Upsilon_{c_{j_\beta}} \tag{3.9}$$

The behavior of the component at any instant in time is nothing but the valuation of the quantities at that time instant. In formal terms, the behavior of a component β with monitored and controlled quantities $\{m_{1_\beta}, .., c_{1_\beta}, ..\}$ (collectively referred as β) at select instances/duration of time T, can be defined as a run (based on equation 3.3),

$$\theta_\beta^T = \{(t, m_{1_\beta}(t), \ldots c_{1_\beta}(t), \ldots)\}_{t \in T} \tag{3.10}$$

Accordingly, the set of all possible runs θ_β^T for all possible T, denoted by Θ_β, represents all the possible behaviors of β.

3.2.2 Requirements

Our interest in designing and constructing a component is to ensure that it exhibits certain desired behaviors in which quantities' values (at least some of them) are constrained. The descriptions of such desired behaviors, typically specified by those who would use the component and/or pay for its development, are called its *requirements*. Typically, the stakeholders of the system express requirements as equations, inequalities and/or predicates over a subset of the component's quantities[1]. However, when interpreting them as desired behaviors of the component, they can be seen as runs of the component in which the values of certain quantities are constrained (per the conjunction of all expressed constraints) while the rest of the variables are unconstrained.

For example, consider a monitoring device that can sense the room temperature (`High or Low`) and produce an buzzing noise (`Sound` or `Silent`), and has a silence button (`Pressed or Not Pressed`). Let us say the device is expected to *Sound the buzzer when a High temperature is sensed.* Interpreting this over time $T = \mathbb{R}^+$, the runs with elements {`t, Temp=High, Button=*, Buzzer=Sound`} and {`t, Temp=Low, Button=*, Buzzer=*`} are expected behaviors, while runs with elements {`t, Temp=High, Button=*, Buzzer=Silent`} are not ($t \in T$ and `*` denotes an unconstrained value).

In general, the requirements of a component (β) with a set of monitored quantities and controlled quantities (as depicted in Figure 3.1), denoted by $\mathcal{R}_\beta$, is defined as a subset of all runs of a component over a certain time,

Figure 3.1: Component

$$\mathcal{R}_\beta \subseteq \Theta_\beta \tag{3.11}$$

[1]While some approaches [88] allow requirements to be expressed using any quantity, we require that requirements are expressed over component's quantities, as recommend in [74].

While such requirements are a set of runs over certain time windows, the actual runs of a component could concern different time instances. However, since our concern is building components whose runs meet the constraints imposed by the requirements, we establish the following relationships between the two sets of runs.

Satisfaction

A component's run is said to *satisfy* the requirements when each element of the component's run meets the constraints of the requirements in the time instances where the requirements are defined. In other words, the satisfaction relationship establishes that the component's run is an extension (or superset) of a requirement's run.

Definition 1 *A run $\theta \in \Theta_\beta$ is said to satisfy the requirements $\mathcal{R}_\beta$, denoted by $\theta \vdash \mathcal{R}_\beta$, if*

$$\exists \theta' \in \mathcal{R}_\beta \quad (\theta' \subseteq \theta) \tag{3.12}$$

Acceptability

While observing runs of a component that *satisfy* the requirement is ideal, it might be a strong constraint (at times even impossible) to achieve in practice. For instance, it is not feasible to assert if a component's run satisfies a requirement specified over a continuous time window, computers operate in discrete time. Hence, a relaxed version of satisfaction in which runs that satisfy the constraints of the requirement during an acceptable set of time instants is often considered sufficient. For instance, when testing a system, one often checks if the requirements are met at a few instances of time within the time specified in the requirements (one runs a limited set of test cases). We call this relaxed version of satisfaction *acceptability*. Intuitively, this relation establishes that the component's run extends to a satisfying run.

Definition 2 *A run, $\theta \in \Theta_\beta$ is said to be an acceptable run, denoted by $\theta \mid\sim \mathcal{R}_\beta$, if*

$$\exists \theta' \in \Theta_\beta \quad \cdot \quad (\theta \subseteq \theta') \wedge (\theta' \vdash \mathcal{R}_\beta) \tag{3.13}$$

Since an acceptable run is extendable to a satisfying run, it is worthy to note that the set of all acceptable runs of a component, Θ_acc_β, includes the set of all satisfying runs Θ_sat_β, i.e,

$$\Theta\_sat_\beta \subseteq \Theta\_acc_\beta \tag{3.14}$$

Consistency

The notion of satisfaction and acceptability are meaningless if there are no possible interpretation of runs for a given set of requirements; in other words, if $\mathcal{R}_\beta = \emptyset$, any run of the component will trivially satisfy it. Hence, in any meaningful definition of requirements it is essential to rule out such situations.

One fundamental reason for "$\mathcal{R}_\beta = \emptyset$" is the presence of logically contradicting constraints defining the runs. For instance, the two requirements: *The buzzer shall produce a sound to indicate that the device is powered on* and *The device shall always be silent during its operation* are contradictory. Consequently, when interpreting these constraints, there will be no runs that meet them simultaneously.

Further, some conflicts can occur at specific situations. For instance, consider two requirements: *The buzzer shall produce a sound if the temperature sensed is High* and *The buzzer shall be silent, if the silence button is Pressed.* Only when the silence button is `Pressed` and the temperature is `High` at the same time will the behaviors described by the requirements conflict. Since the values of monitored quantities are not controlled by the component, it is crucial to ensure that requirement interpretation is conflict free

at every possible valuation of monitored quantities.

We will call this property of requirements to be free of such logical contradictions and conflicts as *consistency*. Mathematically, for every possible combination of monitored quantities values in a component (denoted by Θ_{m_β}), if there is at least one combination of controlled quantities values (in the set of Θ_{c_β}), whose join operation results in an acceptable run, then the requirements can be called consistent.

Definition 3 *A set of requirements, $\mathcal{R}_\beta$, is said to be consistent if there is at least one acceptable[2] run of β for every possible monitored quantitie's runs.*

$$\forall\ \theta^T_{m_\beta} \in \Theta_{m_\beta} \cdot \exists\ \theta^T_{c_\beta} \in \Theta_{c_\beta}\ \cdot\ (\theta^T_{m_\beta} \bowtie \theta^T_{c_\beta}) \mid\sim R_\beta \tag{3.15}$$

Sustainable Consistency

While the above definition of consistency ensures that requirements are free of conflicts and contradictions within the time instances of interest of the requirements, it is not strong enough to avoid certain other problems with requirements such as *clairvoyance* or look-ahead. A clairvoyant requirement is one that specifies constraints on controlled quantities' values based upon future occurrence of monitored quantities' values. Such requirements are problematic in general since it is not possible to develop a solution that predicts all the future occurrences of monitored quantities whose values are not controlled by the solution. Hence, analyzing if the requirements are consistent only within the time instances specified in the requirements is not sufficient for non-clairvoyance.

Let us illustrate this with an example: *The device shall make a buzzing sound before the temperature becomes High.* Say, the requirements are only concerned with time $T = \{1, 2\}$ and for each possible combination

[2]While we use acceptability in this formulation to allow cases where observing satisfaction is not practically possible, acceptability can be replaced by satisfaction if needed without loss of rigor.

of temperature (monitored quantity), a consistent and acceptable run exists. The run: {{1, Temperature=low, Buzzer=Sound},{2, Temperature=High, Buzzer=Silent}} is both an acceptable and consistent run (per 3.13 and 3.15). However, if we extend the run for another second (i.e., allow a device to run longer) such as {{1, Temperature=low, Buzzer=Sound},{2, Temperature=High, Buzzer=Silent}, {3, Temperature=High, Buzzer=Silent}}, the extension invalidates the constraint specified at $t = 2$. Hence, it is desirable that such extensions of acceptable runs do not invalidate the constraints specified by the requirements.

Unfortunately, the aforementioned formulation of consistency (as well as consistency formulation in other reference models) are not strong enough to ensure that all future extensions of a run retain consistency. Hence, we need a stronger property that ensures that consistency is *sustained*.

Sustainability of a run's consistency is a non trivial property to formulate. Intuitively, sustainability means that future extensions of an acceptable run retain its acceptability. However, the complexity in formulating such a property is the need to future extend the run for all possible future valuations of monitored quantities (since any value of monitored quantities can occur in the future).

We can describe sustenance as a *diamond property*, as shown in Figure 3.2. Given an acceptable run, it is first projected to its monitored and controlled quantities. This allows the projections to be independently future extended to a desired time T'. For all possible future extensions of the monitored quantity projection (left side of the diamond), one should be able to join it with at least one possible future extension (to T') of the controlled quantity projection (right side of the diamond), such that the resulting run is acceptable.

Definition 4 *A set of requirements is said to be sustainably consistent if the run resulting by joining any future extensions of the monitored quantities with a corresponding*

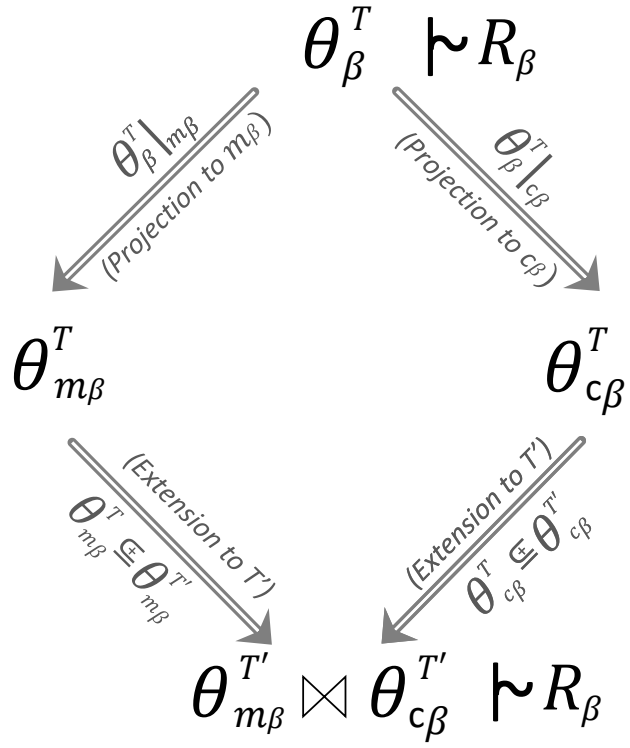


Figure 3.2: Sustainable Consistency

extension of the controlled quantities is acceptable.

$$\forall\, \theta^{T'}_{m_\beta} \cdot \exists\, \theta \cdot (\theta \mathrel{|\!\sim} \mathcal{R}_\beta) \wedge (\theta|_{m_\beta} \sqsubseteq \theta^{T'}_{m_\beta}) \Rightarrow$$
$$\exists\, \theta^{T'}_{c_\beta} \cdot (\theta|_{c_\beta} \sqsubseteq \theta^{T'}_{c_\beta}) \wedge (\theta^{T'}_{m_\beta} \bowtie \theta^{T'}_{c_\beta} \mathrel{|\!\sim} \mathcal{R}_\beta) \tag{3.16}$$

3.2.3 Architecture

Given a set of requirements, designing and developing a component by decomposing it into smaller sub-components such that they compositionally achieve the component's requirements is a common—yet complex—engineering strategy. This primarily involves defining a set of smaller sub-components, interconnecting them, and allocating (or flowing-down) requirements to each of them such that together they achieve the

component's requirements. When performed recursively, this decomposition results is a hierarchy of interconnected components.

Since our interest is to rigorously understand and analyze the requirements that are flowed-down in this process, we define the artifacts in the decomposition and discuss their relationship to requirements. Let us begin by focusing at one level of decomposition of a component at a time—the component and its interconnected sub-components—and call the component the "parent" and the sub-components its "child" components. The interconnection in a decomposition is nothing but a mapping that is established between (and among) the quantities of the parent and its child components; we call this mapping the component's *architecture*.

To uniquely refer to the parent and its child components, we define a simple naming scheme. For a component β with monitored and controlled quantities denoted by $m_{1_\beta} \ldots, c_{1_\beta} \ldots$, let us refer its n sub-components as $\beta.1$,... $\beta.n$ and their respective monitored and controlled quantities as $m_{1_{\beta.1}} \ldots c_{1_{\beta.1}} \ldots, m_{1_{\beta.n}} \ldots c_{1_{\beta.n}} \ldots$ (Figure 3.3).

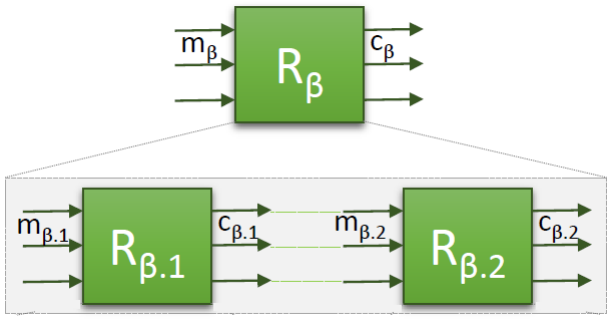


Figure 3.3: Hierarchical Decomposition (One level of the decomposition)

Intuitively, mapping between two quantities allows the values produced by one quantity to be observable by the other. In that sense, we categorize the quantities in a component's architecture that produce the values as *Sources* and those that use (or consume) them as *Destinations*. In a component's (say component β's) architecture,

the parent's monitored and child components' controlled quantities produce values and hence belong to the set of Sources, denoted by $\mathbf{S}_\beta$, whereas the parent's controlled and child components' monitored quantities that receive values from the sources belong to the set of Destinations, denoted by $\mathbf{D}_\beta$. The mapping of sources to destinations is the architecture. Formally,

Definition 5 *The architecture of a component β, denoted by $\mathcal{A}_\beta$ is a subset of the relation between $\mathbf{S}_\beta$ and $\mathbf{D}_\beta$.*

$$\mathcal{A}_\beta \subseteq \{(s,d) |\ \ s \in \mathbf{S}_\beta \ \ ; \ \ d \in \mathbf{D}_\beta\} \tag{3.17}$$

where

$$\mathbf{S}_\beta = \{m_{1_\beta}, m_{2_\beta}, \ldots, c_{1_{\beta.1}}, c_{2_{\beta.1}} \ldots c_{1_{\beta.n}}, c_{2_{\beta.n}} \ldots\}$$

$$\mathbf{D}_\beta = \{c_{1_\beta}, c_{2_\beta}, \ldots, m_{1_{\beta.1}}, m_{2_{\beta.1}} \ldots m_{1_{\beta.n}}, m_{2_{\beta.n}} \ldots\}$$

Since the architecture establishes value sharing, the mappings should be free of errors and uncertainty about determining the shared values at all times. Hence, certain restrictions on these mappings, that we call *architectural constraints*, apply. Namely,

- The quantities mapped must be of the same type[3]

$$\forall (s,d) \in \mathcal{A}_\beta \ \cdot \ \Upsilon_s = \Upsilon_d \tag{3.18}$$

- No quantity can be mapped to multiple sources.

$$\forall (s,d) \in \mathcal{A}_\beta \ \ \neg\exists (z,d) \in \mathcal{A}_\beta) \ \cdot \ z \neq s \tag{3.19}$$

[3]Mapping between compatible types is allowed. However, we intentionally avoid the discussion on type compatibility to keep the reference model simple.

An architecture meeting the above constraints, once established, allows seamless sharing of values between the quantities; in other words, the architecture establishes sharing of values at every instant of time. The *semantics* of such an architecture (how the values are shared) is defined over $\mathbb{R}^+$ (the domain of the quantities) as:

$$\forall (s, d) \in \mathcal{A}_\beta, \forall t \in \mathbb{R}^+ \ \cdot \ s(t) = d(t) \tag{3.20}$$

By defining the architecture this way, we intentionally, allow quantities to be left without any mapping. This is because in practical middle-out developments, several quantities may be irrelevant to achieve the overall requirements. So, the designers may choose to leave those quantities (or ports) unconnected. This reference model leaves the decision of connections to the designer and does not enforce completeness.

Composition Realizability

Despite each component's requirements being consistent and its architecture respecting the architectural constraints, the act of putting these components together may introduce conflicts. For instance, consider a simple requirement: *When input a is true, the system shall set output b to false*; If one maps the output b to the input a (a feedback loop) during the architecture phase, it will result in a logical contradiction. To avoid such architecture induced issues, we define a property called *Composition Realizability* that establishes the joint consistency of requirements and architecture; in other words, this property establishes that it is possible to realize the composition.

Definition 6 *The architecture and requirements are said to the jointly realizable, if there exists at least one satisfying run for each of the sub-components for every monitored quantity value of its (parent) component that are composable per the architecture.*

$$\forall\ \theta_{m_\beta} \cdot \exists\ \theta_1, \theta_2 ... \cdot\ (\theta_1 \vdash \mathcal{R}_{\beta.1})\ \wedge\ (\theta_2 \vdash \mathcal{R}_{\beta.2}) \cdots \wedge \mathcal{A}_\beta \tag{3.21}$$

where $\theta_{m_\beta} \in \Theta_{m_\beta}, \theta_1 \in \Theta^{T'}_{\beta.1},\ \ \theta_2 \in \Theta^{T''}_{\beta.2}, \ldots \theta_n \in \Theta^{T'''}_{\beta.n}$

Hierarchical Requirements Satisfaction/Acceptance

Given a set of component requirements, its sub-component requirements, and their architecture, the next step is to ensure that the components and their requirements composed as per the architecture indeed *satisfy* the component's requirements. To do so, we need to establish that if every run of each component (both parent and child components) in the architecture satisfies their respective requirements, given the architecture, we can assert that the component's requirements are *hierarchically satisfied*.

Definition 7 *Component requirements are said to be hierarchically satisfiable, if every run of the component and sub-component is a satisfying run of its respective component/sub-component in the given architecture.*

$$\forall \theta_1, \theta_2, \ldots \theta_n\ , \theta\ \ \cdot\ \ (\theta_1 \vdash \mathcal{R}_{\beta.1})\ \wedge\ (\theta_2 \vdash \mathcal{R}_{\beta.2}) \cdots \wedge \mathcal{A}_\beta \Rightarrow \theta \vdash \mathcal{R}_\beta \tag{3.22}$$

where $\theta \in \Theta^T_\beta$ and $\theta_1 \in \Theta^{T'}_{\beta.1},\ \ \theta_2 \in \Theta^{T''}_{\beta.2}, \ldots \theta_n \in \Theta^{T'''}_{\beta.n}$

In recursively decomposed systems, establishing these rules at every level helps precisely assure that the system meets the user's needs.

3.3 Discussion

The proposed approach to structuring requirements based on the systems' architecture is a shift from the view of requirements suggested by the existing models. When the requirements are organized/conceptualized based on this model, the top-level requirements (or overall needs) would be at the highest level of the hierarchy; at the next level, the components that collaboratively satisfy those requirements, namely, the system to-be-developed and the part of the environment that is necessary to satisfy those user needs would be structured as interacting (child) components. The further decomposition of the system-to-be-developed would be organized in subsequent levels.

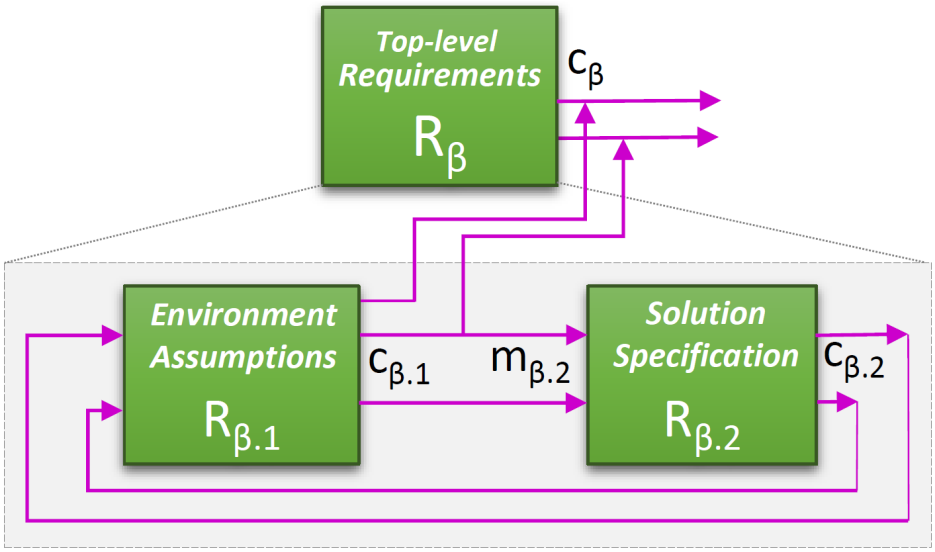


Figure 3.4: Top level in the hierarchy

The organization scheme in which the top-level requirements at the parent level and the environment-system as collaborating child components, as shown in Figure 3.4, is a significant change in perspective when compared to other models. This proposed view highlights that the environment and system have an intertwined relationship in satisfying the requirements. Though existing models discuss the collaborative satisfaction

by the system and environment, they either rely on humans to abstractly define "what is present" verses "what needs to be developed", or use the grammatical moods (indicative and optative) of expressing the descriptions as a basis to distinguish them [45]. However, in complex, middle-out developments where the description of several parts at several levels of abstraction are often present, this distinction become challenging. Whereas this proposed way of structuring provides a natural platform for engineers to distinguish requirements of each component and handle their appropriation at every level of system abstraction.

Further, the uniform component-oriented approach serves as a guidance to progressively refine requirements. Though similar guidance for scoping requirements artifacts is discussed in the models such as the Four Variable model, they naturally relate only to artifacts specified at higher-levels of system detail; whereas, when discussing systems and its components with complex, hierarchical architectures, explicit accounting of the intertwined relationship between requirements-architecture becomes inevitable.

3.3.1 Comparing Models

The WRSPM and Four Variable models are undoubtedly forerunners in discussing the requirements of complex system in an abstract yet formal manner. In this section, we compare and contrast our model with these two related models. While a one-to-one correspondence to these models may not be obvious due to some differences in their conceptual views and formal definitions, it is not difficult to establish that these existing models are specific instances of the hierarchical reference model.

A fundamental difference is that the Four Variable and WRSPM models are based on a flat, fixed architecture (i.e., separation among components) and shared variables between them, whereas our model defines a hierarchical, flexible way of defining components and their architecture. Another difference is the way the artifacts are defined.

They are treated as relations or predicates among phenomenon/variables in the other models whereas we define them as runs. Further, the notion of time is either abstracted away or not adequately handled in the existing models, that had lead to practical ramifications [34]. On the other hand, we have defined runs that inherently includes time and have captured unique properties such as acceptability and sustainable consistency that time. In the following discussion, we elaborate these points.

Four Variable Model vs Hierarchical Model

Recollecting from Chapter 2, the Four Variable model defines four distinct collections of variables: m for monitored values, c for system-controlled values, i for values input to the program's registers, and o for values written to the program's output registers. There are also five predicates formally representing the necessary documentation: NAT(m, c) describing nature without making any assumptions about the system, REQ(m, c) describing the desired system behavior, IN(m, i) relating the monitored real-world values to their corresponding internal representation, OUT(o, c) relating the software-generated outputs to external system-controlled values, and SOF(i, o) relating program inputs to program outputs.

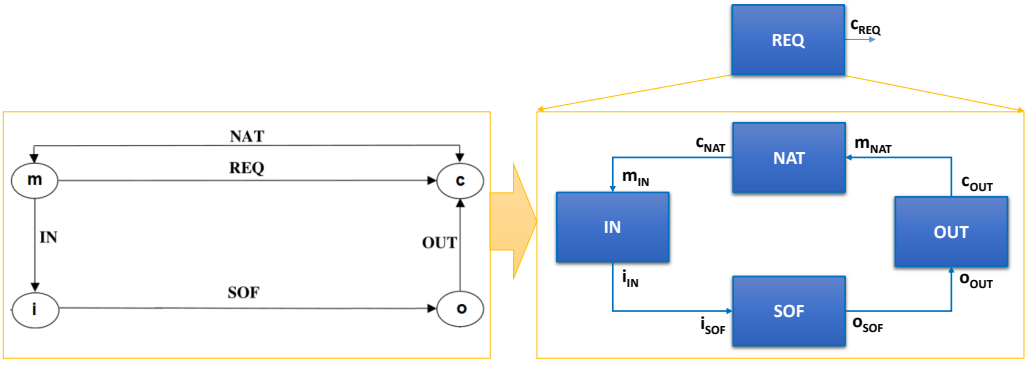


Figure 3.5: Four Variable Model in terms of Hierarchical Model

Recasting this model in terms of our reference model, each of REQ, NAT, IN, OUT

and SOF becomes a component with a unique set of quantities, as shown in Figure 3.5, and an architecture that connects them. Since, we are moving from a flat to an hierarchical pattern when establishing the architecture, one would define REQ at the highest level whereas NAT, IN, SOF and OUT would be defined as interacting components such that they (together) satisfy the higher level REQ.

Let us discuss now the consequences of this recasting. As mentioned earlier, structuring REQ at a higher level in the hierarchy than NAT is a shift from the the traditional view of both at the same level of detail. While one could argue that the traditional view supports the idea that requirements are specified in-terms of the environment quantities, this new view does it even more pragmatically. As discussed in the Twin Peaks model, engineers progressively traverse between "what is already available" (NAT) and "what can be developed in practice" (IN-SOF-OUT) to determine what can be practically be achieved (REQ). Hence, it is difficult to define requirements purely in terms of existing quantities in the environment; in fact, several behaviours that one desires as requirements may not even exist in the current environment and only gets introduced as we explore a solution. Hence, it is conceptually straightforward to conceptualize NAT-IN-SOF-OUT as interacting components to meet REQ. While the traditional recommendation to express requirements in-terms of the environment quantities holds wells in this model since the architecture allows the quantities between NAT and REQ to be mapped, this model brings to light that appropriate environment quantities are discovered and evolved as one explores the Twin Peaks.

Beyond the conceptual structure, there are three major proof obligations defined in the Four Variable model. The first set of properties are called feasibility and requires:

$$\forall m \cdot (\exists c \cdot NAT(m, c)) \Rightarrow (\exists i \cdot IN(m, i))$$

$$\forall m \cdot (\exists c \cdot NAT(m, c)) \Rightarrow (\exists c \cdot REQ(m, c))$$

The former obligation requires that IN must handle all possible values of monitored quantities under NAT. The latter property requires that for every value of the input quantity that occurs in the environment, if there is an environmental behaviour, then there should be at-least one behaviour defined in the requirement.

These obligations are consequences of the consistency properties and architecture definition in our model. While the Four Variable model defines REQ, NAT, IN, SOF and OUT as relations expressed over quantities, our reference model considers them as runs. Formally, runs are nothing but interpretations that meet the constraints of the relations. Hence, in terms of our reference model, the obligations of NAT and IN can be defined using the architectural mapping and consistency properties as follows:

$$c_{NAT} = m_{IN}$$

$$\forall\ \theta_{m_{NAT}} \in \Theta_{m_{NAT}} \cdot \exists\ \theta^T_{c_{NAT}} \in \Theta_{c_{NAT}}\ \cdot\ (\theta^T_{m_{NAT}} \bowtie \theta^T_{c_{NAT}}) \mid\sim NAT$$

$$\forall\ \theta^T_{m_{IN}} \in \Theta_{m_{IN}} \cdot \exists\ \theta^T_{c_{IN}} \in \Theta_{c_{IN}}\ \cdot\ (\theta^T_{m_{IN}} \bowtie \theta^T_{c_{IN}}) \mid\sim IN$$

These together establish that for every value of NAT's monitored quantity there is at least one behaviour described in NAT, at every instant in time the value produced at NAT is mapped to monitored quantities of IN and for every value of monitored quantity of IN there is at-least one acceptable behaviour described in IN.

The latter feasibility property of the Four Variable model relates REQ and NAT. REQ in our model is at the highest level and hence will have behaviours described in-terms of controlled quantities. Hence, this obligation simplifies to

$$\exists\ \theta^T_{c_{REQ}} \in \Theta_{c_{REQ}}\ \cdot\ (\theta^T_{c_{REQ}}) \mid\sim REQ$$

$$c_{REQ} = c_{NAT} \cup c_{OUT}$$

The above properties together establish that for every desired behaviour described in REQ is also a behaviour of NAT ($c_{OUT} = m_{NAT}$). Hence, the two feasibility obligations specified in the Four Variable model can also be established in our model.

The third obligation of the Four Variable model is the acceptability obligation,

$$\forall m, i, o, c \cdot NAT(m, c) \wedge IN(m, i) \wedge SOF(i, o) \wedge OUT(o, c) \Rightarrow REQ(m, c)$$

This is exactly the hierarchical satisfaction property of the Hierarchical Reference model defined in 3.22, except that the above uses a relational formulation whereas we use runs (that are interpretations of the relations). In other words, the acceptability obligation in our formulation is:

$$\forall\ \theta_1, \theta_2, \theta_3, \theta_4, \theta \cdot (\theta_1 \mid\sim NAT) \wedge (\theta_2 \mid\sim IN) \wedge (\theta_3 \mid\sim SOF) \wedge (\theta_4 \mid\sim OUT) \wedge \mathcal{A} \Rightarrow \theta \mid\sim REQ \tag{3.23}$$

where $\theta \in \Theta^{T}_{REQ}$, $\theta_1 \in \Theta^{T'}_{NAT}$, $\theta_2 \in \Theta^{T''}_{IN}$, $\theta_3 \in \Theta^{T''}_{SOF}$, $\theta_4 \in \Theta^{T'''}_{OUT}$

The benefit of interpreting requirements as runs in our reference model helps inherent include the notion of time and reason about unique properties involving it. Though the Four Variable model briefly mentions that the values of variables are functions of time, the challenges with interpreting the relations/predicates with the notion of time have not been discussed. For instance, one would not be able to reason about properties involving time such as acceptability and clairvoyance using the Four Variable model's formulation. Further, the model is not equipped to define complex, hierarchical architectures and the resulting system behaviours. This makes it practically challenging to understand the modern, complex systems in practice. On the contrary, our model inherently includes time in the definitions that has lead to defining a unique properties that can not be

defined in terms of the other models.

WRSPM Model vs Hierarchical Model

Recollect from Chapter 2, the WRSPM model is based on the widely accepted separation between the system and its environment, a number of key artifacts which pertain to the two, and a vocabulary, which is used to describe the environment, the system and their interface. The key artifacts are: the domain knowledge, W, is what we know about the World (aka environment); the requirement, R, is what the customer requires of a system working within the environment; The specification, S, is a description of the system, which can be used for its implementation; The program, P, is the system implementation; and, the programming platform, M, is the machine within which the program runs.

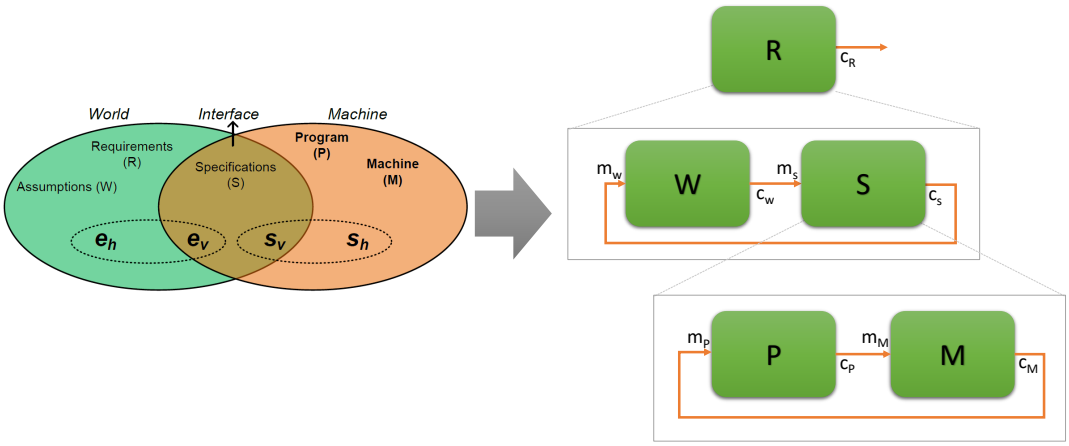


Figure 3.6: WRSPM Model in terms of Hierarchical Model

In terms of our reference model, one can define a component for each of W, R, S, P and M with a unique set of quantities for each and establishing the variable sharing using the architecture definition as illustrated in Figure 3.6. While establishing the architecture, R is defined at the highest level, W and S as interacting components at

the next level, and P and M as interacting set of component decomposed from S at the subsequent level. This way, the architectural structure/separation defined in the WRSPM model can be defined in our reference model. On the contrary, there is no provision in the WRSPM model to capture any other complex interaction, decomposition or hierarchical architectural pattern.

The WRSPM model also defines formal obligations similar to the Four Variable model. However, a major difference is lack of notion of time in this model. While we defer to the implications of abstracting away time to later in this chapter, for now we compare the formulations by conceptualizing the time (T) for every component is 1. The first property defined in the WRSPM model is called consistency of the domain knowledge:

$$\exists e, s \cdot W$$

Intuitively, this property states that the (description of the) the World exists and is consistent. Comparing this formulation with our consistency property formulation (3) for the W component, our formulation is stronger. Our consistency formulation requires the description of W to be consistent for all possible valuation of its monitored quantities (i.e., controlled quantities of S); the same applies to the S component. While this might superficially appear as a stronger condition, this brings to light the Twin Peaks between the description of World and the machine. In other words, we require that the machine description accounts for all the possible values of quantities it monitors from the World, as well as that the World description accounts for all the possible values of quantities the machine produces and posed to the World. For the R component at the top-most level in our hierarchy, this consistency formulation is simplified to an existential quantifier similar to WRSPM's consistency formulation.

The second obligation defined in WRSPM is *Adequacy* :

$$\forall e, s \cdot W \wedge S \Rightarrow R$$

This embodies the proof obligation that one is specifying the right system, i.e., one that satisfies the requirements. Similar to the Four Variable model's acceptability property, the WRSPM's adequacy property naturally corresponds to the hierarchical satisfaction property of our hierarchical reference Model defined in Definition 3.22.

The last obligation defined in the WRSPM model is Relative consistency:

$$\forall e_v \cdot (\exists e_h, s \cdot W) \Rightarrow (\exists s \cdot S) \wedge \forall s \cdot (S \Rightarrow \exists e_h \cdot W)$$

This is a unique, significant contribution of this model that merits some discussion. Intuitively this property establishes that S (description of desired machine behaviours) must account for all possible World phenomenon that can be "monitored" by the machine and such machine behaviours are all consistent with W (description of the World). This last condition has profound implications on the nature of the relationship between the World and the Machine. It is meant to establish that the machine's behaviour is complete and consistent with respect to the World.

The consistency property formulation defined in our reference model is influenced by this property. While we include all values of monitored quantities in the quantification in the consistency formulation (Definition 3), we do not include the World component (such as W). This is intentional since it is almost always the case that there is co-evolution of the world and the machine in complex, middle-out developments. Hence, one would not be able to fix a world definition when refining the machine specification. Rather, one has to define a machine given the possible inputs it can receive from its World, as well as impose additional constraints or expectations to the World to accommodate the new machine such that they cooperatively meet the overall goal (hierarchical satisfaction). Hence, our consistency formulation for W and S together

achieve the purpose of relative consistency.

$$c_W = m_S; \quad c_S = m_W$$

$$\forall\, \theta_{m_S} \in \Theta_{m_S} \cdot \exists\, \theta_{c_S} \in \Theta_{c_S} \;\cdot\; (\theta_{m_S} \bowtie \theta_{c_S}) \mid\sim S$$

$$\forall\, \theta_{m_W} \in \Theta_{m_W} \cdot \exists\, \theta_{c_W} \in \Theta_{c_W} \;\cdot\; (\theta_{m_W} \bowtie \theta_{c_W}) \mid\sim W$$

Overall, the two existing models are both conceptually and formally specific instances of the hierarchical model. Beyond the properties discussed in both models, the significant contributions of our reference model are the two properties—sustainable consistency and decomposition realizability. Establishing the sustainable consistency property helps identify inconsistencies due to the way time is handled in requirements, such as the problem of accidentally requiring clairvoyance. Establishing the decomposition realizability property helps ensure architecture induced inconsistencies do not occur when decomposing systems. As engineers hierarchically decompose systems and progressively define requirements, establishing these properties at each level helps ensure the correctness of the decomposition and the requirements early in the development phase. We believe that our way of interpreting requirements as runs and hierarchically intertwining architecture with requirements has allowed us to elegantly formulate these properties.

3.4 Summary

In sum, this hierarchical requirements reference model provides a framework to structure requirements along the architectural lines of the system. While several fundamental concepts and properties in this model are influenced by the existing references

models and approaches such as KAOS [84], SPEEDS [20] and other contract-based approaches [13, 88] in addition to the Four Variable and WRSPM models, those concepts are in our hierarchical model generalized in a way that allows engineers to use this model as a reference to deal with system requirements at any level of abstraction with any complexity. In fact, one can envision existing models as instances of the Hierarchical Reference Model with specific decomposition schemes and use the properties defined in this model as guidance to systematically approach the Twin Peaks. In later chapters, we describe how the rules and concepts defined in this model help guide requirements analysis of a complex system as well as influence defining novel techniques to analyze requirements in practice.

Chapter 4

Case Study

The only time you can really find the best problem definition is after you found the solution. - De Bono [14]

In this dissertation, we use a generic infusion pump system as an illustrative example. The choice of an infusion pump for our research was motivated by the collaborative effort by the system engineering community and the U.S. Food and Drug Administration's (FDA) initiative [25] to explore the root cause of various system and software errors that lead to catastrophic incidents and safety recalls of the pumps [26]. The goal of that effort was to provide guidance to device manufacturers to improve the safety of these devices. One of the areas explored to demonstrate the safety of an infusion pump developed is proving that the device satisfies a set of requirements that were defined to mitigate the hazards [37, 52]. Our aim is to contribute to this initiative by providing an archetype of requirements engineering artifacts for a Generic Infusion Pump and demonstrate with the help of novel guidance/approaches that a device that satisfies those requirements is indeed safe. Such exemplar artifacts and approaches, we believe, shall serve as a reference for researchers, practitioners, and authorities who develop and certify such systems.

Further, the generic infusion pump serves as a suitable, non-trivial case example to

investigate and illustrate our research goals. Typically, commercial infusion pump systems are developed by iteratively composing COTS products, integrating with existing sub-systems, reusing components used in previous pump models, and newly developed components. Consequently, middle-out composition and hierarchical requirements flow-down is inherent in their development. That said, an infusion pump serves as an apt case example to illustrate the practicality of our reference model to engineer requirements.

4.1 Infusion Pump Overview

Infusion pumps are medical cyber-physical systems used for controlled delivery of liquid drugs into a patient's body according to a physician's prescription, a set of instructions that governs the plan of care for that individual. These pumps may be classified into various kinds depending on their features, construction, and usage. Patient-controlled analgesia (PCA) pumps are generally equipped with a feature that allows patients to self-administer a controlled amount of drug, typically a pain medication.

These pumps typically provide multiple modes of drug delivery. In *basal* mode, the drug is delivered at a constant (and usually low) rate for an extended period of time. In a *bolus* mode, the drug is delivered at a higher rate for a short duration of time to address some immediate need or to increase the drug delivery according to some therapy regimen. There may be multiple bolus modes. In *clinician bolus* mode, the drug is delivered at an elevated rate in response to a clinician's request. For example, the clinician may prescribe an elevated rate of infusion for a period of time at the beginning of infusion therapy. Further, in a PCA system, a *patient bolus* mode may be activated to deliver additional drug in response to a patient's request for more medication, typically to alleviate acute pain. Further, when the device is paused for a short duration due to some medical/safety reason, the device is often allowed to continue delivering the drug at a very low intravenous drip rate called the *Keep Vein Open*(KVO) to prevent the

vein from closing.

4.1.1 Generic Patient Controlled Analgesic (GPCA) Pump

Figure 4.1 shows an external intravenous Generic Patient Controlled Analgesia (GPCA) device in a typical usage environment, a hospital or a clinic. We refer to the device along with its environment as infusion system and the device in isolation as GPCA system. In an infusion system, the clinician operates the GPCA device, programs the prescription information, loads the drug, connects the device with the patient, and responds to exceptional conditions that occur during the therapy. The patient receives the medication from the device through an intravenous needle. The patient can self administer prescribed amounts of additional drug by requesting a bolus, a request usually done by pressing a bolus request button accessible at the patient's bed. The hospital pharmacy database is a repository that stores manufacturer provided drug information such as ranges of values for various drug parameters that are safe for infusion therapy.

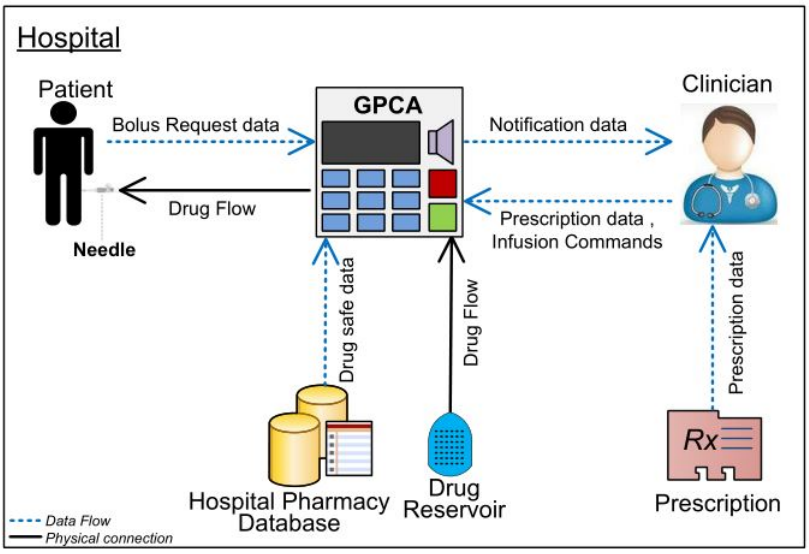


Figure 4.1: Generic Patient Controlled Analgesic Infusion Pump

The GPCA system has an interface to this repository for accessing this drug data that it used for verifying various infusion parameter values against the drug specific data from the repository to ensure that the programmed therapy regimen is within safe limits. The GPCA system has three primary functions (1) deliver the drug based on the prescribed schedule and patient requests, (2) prevent hazards that may arise during its usage, and (3) monitor and notify the clinician of any exceptional conditions encountered.

4.1.2 Integrated Clinical Environment

To understand the decomposition and flow-down of requirements in the development of an infusion pump, we considered a closed loop clinical medical environment in which GPCA is networked with other devices. Several medical studies have revealed that one of the reasons for hazards posed by infusion pumps is their isolated operation without knowledge of the patients' health and other safety mechanisms [6]. For instance, opioid medications have the potential to can cause respiratory failure in patients in certain scenarios depending upon their medical condition [71]. While a properly programmed PCA system should ideally avoid overdoses because it is programmed with upper limits on doses it should deliver, several overdose hazards with PCA have been reported in clinical practice [69]. Hence, to enhance patient safety, infusion pumps are being increasingly considered to communicate with other devices that monitor patient vitals and automatically control the amount of drug delivered to patients. The clinical scenarios provided a high-level context and scope to define safety requirements of the system and the inter-operating devices.

We considered a networked medical system that continuously monitor patients for deteriorating respiratory functions and automatically stop drug infusion if needed, as shown in Figure 4.2. The overall system architecture has a *GPCA pump* that delivered

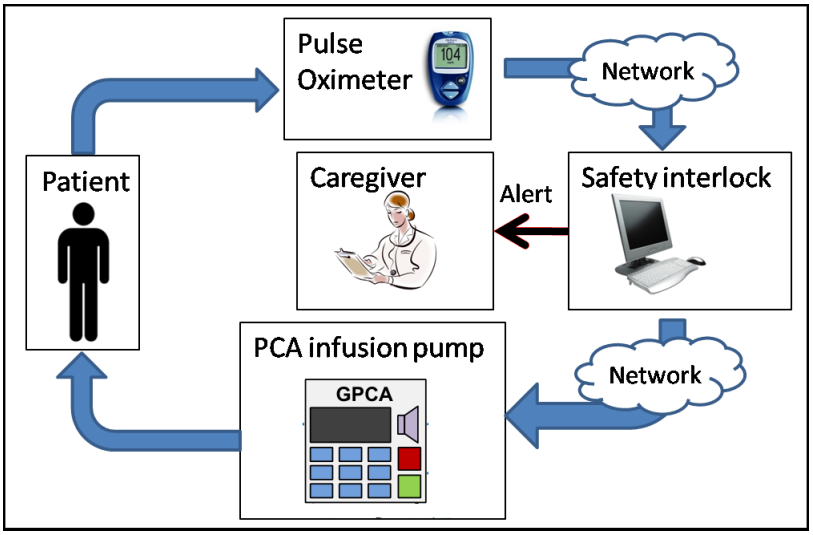


Figure 4.2: GPCA in Closed Loop System

drug to the patient based on a prescription that is programmed by the caregiver, a *pulse oximeter* device that receives physiological signals from a clip on the patient's finger and processes them to calculate heart rate and measure of blood oxygenation (SpO2), a *safety interlock* that interrupts GPCA's infusion and alerts the caregiver if the SpO2 is greater than a certain threshold.

4.1.3 Prior Research and Infusion Pump Artifacts

The FDA had evaluated a broad spectrum of infusion pumps across manufacturers and has encountered problems with device software, human factors, reliability and manufacturing. Based on the evaluation, FDA concluded that many adverse events may be avoided by improving the design verification and validation processes for these devices [24]. To that end, several investigators around the world, in parallel, explored various aspects of the improving the safety of infusion pumps focusing on its software [22, 51], user-interface [58], assurance [1] etc. In fact, researchers at the FDA

launched a Generic Infusion Pump (GIP) project to help address safety problems associated with infusion pump software. They released artifacts such as preliminary hazard analysis, safety requirements, and reference behavioural model of a generic infusion pump to the public [46]. However, to the best of our knowledge, none of these research explores the challenges related to hierarchically specifying and verifying the requirements of infusion pumps that are developed in a middle-out way in practice.

4.2 Requirements Engineering Challenges with the GPCA

Our interest with the GPCA is to explore its requirements and identify the related challenges. The existing, publicly available requirements documents for the GPCA system did not meet our standards of completeness, consistency, and rigor needed to serve as a basis to show that the GPCA is safe. Hence, we focused our efforts to define a suitable set of requirements for the GPCA. Initially, given the available information about the GPCA, we anticipated that this effort would be straight forward and completed in short order. During the process, however, we faced unanticipated challenges that we believe are not limited to the GPCA but also to a wide variety of systems in the safety critical domain. In particular, we faced the following challenges:

4.2.1 Scoping Challenge

In the GPCA project, precisely scoping the system and its environment was far more challenging than we first anticipated. This is partly because the existing documents that stated requirements over the GPCA system had requirements expressed over multiple scopes and there was little guidance to determine which scope would be appropriate. Providing a consistent scope for the requirements is essential to maintain intellectual control of the the development and assurance efforts. We do not want our efforts to get bogged down in implementation details, such as how to operate a particular pump, too

early during development. On the other hand, we do not want the scope of the system to grow as development progresses. For example, consider the following statements from an existing requirements document:

> *"An air-in-line alarm shall be triggered if air bubbles larger than 5 ml are infused into the patient."*
>
> *"When the option to suspend the pump is selected, the current pump stroke shall be completed prior to suspending the pump."*

The above statements are clearly requirements about the GPCA system expressed over different scopes. While the former concerns the entire infusion system interacting directly with the patient, disregarding the internal components of the system, the latter is about a specific component (the physical pump) within the GPCA system in response to an user event. These two statements that express constraints over two very different scopes being a part of the same requirements document causes ambiguity in what needs to be developed. In the GPCA project, these tasks turned out to be quite complicated and time consuming, indicating a strong need for additional methodological support.

While the existing reference models discuss scoping of requirements by delineating the system from its environment and identifying the interface, they do not provide guidance for determining an interface for middle-out and hierarchical construction where components at multiple levels of system detail exists. In the GPCA, we considered several (sub) components to exist and hence there was a need to recursively decompose the system and flow-down requirements that was very challenging without some sort of conceptual guidance.

4.2.2 Traversing the Twin Peaks

As we recursively decomposed the GPCA to delineate the existing components from those needs to be developed, we found that requirements flown-down from a component to its sub-components was tightly co-related to that component's decomposition

or architecture. Any changes to the architecture had a major influence on the requirements. For instance, when we decided to define a new sub-component of the software that would book-keep the parameters, we had to redefine the requirements of a bunch of other sub-components that needed those parameters. Given that few components and its requirements pre-exists in the GPCA, we had to meticulously determine the architectural connections and requirements allocation to the other components.

Further, as we progressively decomposed the GPCA while accounting for existing components, we had to carefully flow-up the limitations and constraints of those components. For instance, when a specific sensor was chosen to meet the high reliability requirement of the GPCA, we had to alter the overall voltage and power requirement of the GPCA to accommodate that sensor's higher power consumption. Similarly, we had to appropriately alter a sub-component's requirement of the software, that calculates the battery dissipation rate of the GPCA and notifies the caregiver if the remaining charge is lower than a certain threshold. Hence, we found that requirements had to systematically flown-up, down and across as the GPCA was decomposed. We believe that without some sort of guidance meticulously performing this iterative, intertwined requirements-architecture refinement was intellectually challenging.

4.2.3 Hierarchical Verification of Requirements

To ensure if requirements of the GPCA and its components were specified correctly and if they collectively meet the overall requirements was a challenge due to the abstraction induced differences among requirements at each level of the GPCA's decomposition hierarchy and the unique concerns addressed by those requirements. As we decomposed the GPCA hierarchically, we found that at different levels of the system hierarchy, the requirements had to account for unique concerns. For instance, the continuous timing concerns had to be accounted when specifying requirements for the clinical scenario; the

probabilistic behaviour of the physical components had to be accounted when specifying requirements of the GPCA's components; the requirements of the software and its sub-components had to take into account its discrete-time behaviour. Also, with progressive levels of system abstraction (detail), the requirements had also had to be appropriately refined. Hence, to reason about the requirements we had to reconcile the abstraction induced differences between the layers and verify the requirements. Unfortunately, none of the existing RE guidance or approaches helped seamlessly reason about requirements that are flown-down in complex, hierarchical system architectures.

4.2.4 Tracing Requirements

As we recursively decomposed requirements of the GPCA, it was not easy to keep track of how the requirements were flow-down to multiple levels of system detail, *requirements traceability*. Nevertheless, Such traceability is crucial to understand how the requirements are met by the system and ensure if the requirements were adequately specified. In the GPCA, In addition to the need for hierarchical traces between requirements, we also found that such traceability is not limited to the requirements at each level but also the architectural detail at each level that is necessary for its realization. In fact, given the existing GPCA artifacts with collection of requirements statements relating to multiple levels of system, the need to systematically record traceability was inevitable. Unfortunately, without a fundamental guidance to relate the requirements-architecture in the RE literature or discussion about assessing requirements for correctness and completeness embracing the architecture, it was challenging to rigorously perform traceability or evaluate the quality of the requirements.

In the following chapters, we discuss how we address the aforementioned challenges using our reference model as a guide.

Chapter 5

Requirements Scoping and Flow-down

In theory there is no difference between theory and practice. In practice, there is. – Yogi Berra

Although requirements engineering is a relatively mature discipline which has improved over decades of research and best practices, it still remains as one of the most challenging phases of development. As discussed in the previous chapter, in modern hierarchic, middle-out systems such as GPCA where several levels of detail at which requirements needs to be captured, distributed and analyzed, the challenges are exacerbated. To cope with such challenges, the hierarchical reference model proposed in this dissertation serves as a helpful frame of reference for engineers to deal with such engineering challenges.

To aid in the exploring, understanding and co-evolving the requirements and architecture of GPCA (the Twin Peaks concept), we used a model-based approach. In this chapter, as we outline the reference-model guided requirements engineering approach of the GPCA, we also describe our experiences, challenges, and lessons learned in the process of formulating and analyzing the requirements and modeling the system

architecture.

The purpose of the research described in this chapter is not to prescribe any specific approach or methodology for describing the type of systems that we are concerned with; instead, we aim to bring to light some of the essential elements and generic strategies that engineers may encounter and consider when dealing with complex system requirements. The choice of approaches that we used for the GPCA entirely depended on the domain and the nature of analysis we intended to demonstrate and contribute to the community. We believe that the development artifacts and techniques presented could serve as a generic reference to be used by researchers, practitioners, and authorities while developing and evaluating complex systems. In the remainder of the chapter, we begin our discussion explaining the role of models in requirements engineering and then discuss our approach to reference model guided, model-based requirements engineering of GPCA.

5.1 Model-based Approach to Requirements Engineering

Models have proved useful during requirements elicitation [92], analysis and validation [81], as well as verification [61] of complex safety related control systems. Hence, we choose a model based approach to requirements engineering of the GPCA since it allows the unambiguous capture of requirements, architecture, and design detail, and enables the extensive use of tools and automation. This allows for greatly reduced manual effort while maintaining or improving the capability of fault finding and enhancing quality of the system [63, 64].

To ascertain that a system is safe and effective, it is imperative to precisely specify its requirements. In practice, however, the requirement are rarely (if ever) well known at the onset of a project. Instead, the initial set of requirements form a basis for a proposed solution (an initial systems architecture or design). In our case, we advocate

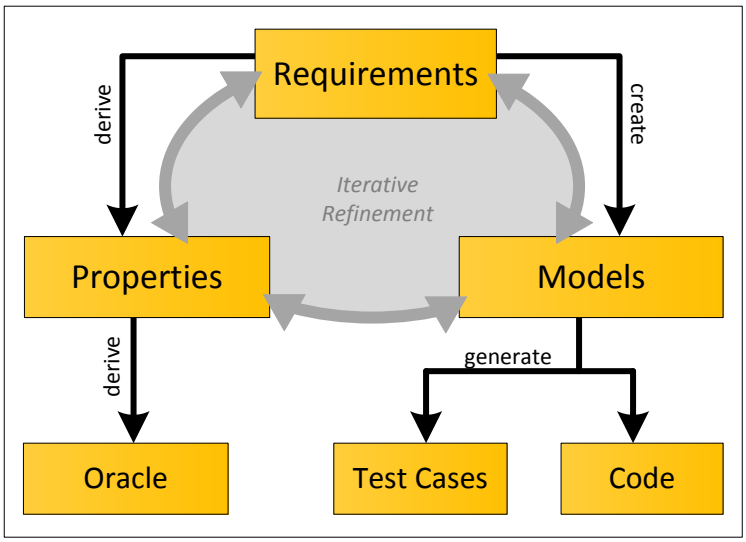


Figure 5.1: Approach Overview

modeling to evaluate design alternatives and explore the desired system. In effect, the models are proposed solutions to the problem at hand (the requirements); the models are early prototypes of a proposed system that help visualize and analyze the problem as well as the proposed solution.

Models and requirements exist in a symbiotic relationship; through iterative evolution they contribute to each other's improvement. While constructing models of a system—by exploring the solution domain—an engineer adds architectural and design detail not specifically stated in requirements, architectural and design information that helps clarify existing requirement as well as discover missing ones [39, 90]. Similarly, as requirements are modified and added, the models evolve to accommodate the new constraints. Throughout the process, while modeling, one may find that the design cannot meet the system-level requirements (for example, the requirements may be unrealizable or there is no cost effective solution). This may lead to the imposition of constraints

on the system's operational environment or a renegotiation of the system-level requirements. Nuseibeh identified this virtuous cycle as the Twin Peaks model [70]. If the modeling notation is formal, various verification techniques can be used to determine if formalized requirements (i.e.,, desirable model properties) are satisfied in the model; a failed verification (the model does not meet the requirements) points to a problem that must be addressed through a modified model, modified requirements, or a further constrained operating environment. Miller et al. first advocated such a process in the context of developing critical avionics systems [63].

In addition to the advantages during the early development phases, a formal model-based approach can be leveraged throughout the life-cycle. Using commercial tools, the models and requirements can be used for code generation, test case generation, and used as oracles during testing. Hence, using a model based requirements analysis approach helped us by providing an insight into the solution domain of the GPCA (model) and capture precise requirements. In the following sections we will discuss the various steps in the process and share the experiences we gained.

5.2 Hierarchical Scoping

Providing a well defined scope for the requirements is essential to maintain intellectual control of the the development efforts. For example, for the GPCA system, if we write requirements in terms of the *prescribed dosage* of a drug, we can postpone discussion of entry errors (the clinician entering the prescribed dosage into the pump is part of our system). On the other hand, from a development perspective, it is most likely preferable to write infusion pump requirements in terms of the *dosage entered at the pump interface*; the clinician is not part of the pump system and entry errors are part of its environment. Thus, the choice of scope does not only have implications for our development efforts (what is "inside" our system and what is "outside" our system),

but also has profound implications for the way the system is assured for safety.

Given that GPCA is a compositional system with components ranging over multiple levels of abstraction, it is even more crucial to demarcate the the boundaries between the environment, the system, and its components even before we attempt to specify requirements.

To that end, our hierarchical model served as a frame of reference to help think of the overall GPCA as a hierarchically composed components with distinct monitored and controlled quantities for each component and sub-component. The need to establish the architecture per the reference model in a hierarchical manner helped precisely determine the scope of each component. Consequently, we were able to explore and express requirements of each component in terms of its quantities in a hierarchic manner. As we explore requirements, to adapt to the limitations of the component, we may readjust the architecture. Hence, for the GPCA, we defined the architecture and requirements in a progressively increasingly level of detail in an alternating manner.

To assist with visualizing and capturing the architecture and the scoping the GPCA system, we relied on architectural modeling.

5.2.1 Architecture Modeling

An architectural model captures how a system is decomposed into a set of interacting subsystems. Engineers use this modeling approach to envision the structure and scope of the system, understand the trade-offs and help focus on the interfaces along with their interactions while abstracting away minute implementation details of each component.

In the GPCA project, we used architectural models to define the components such as the sensors, data entry interface, software, etc and their interaction, as shown in Figure 5.3. As we iteratively decomposed each component, we used the architecture model to capture the elaboration of each component into a set of interacting sub-components.

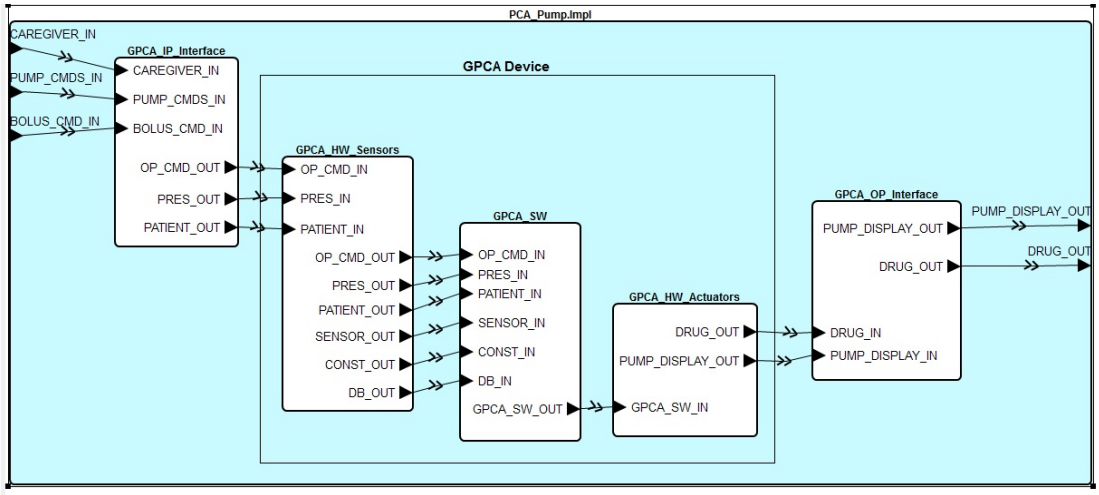


Figure 5.2: GPCA System-level Architecture View

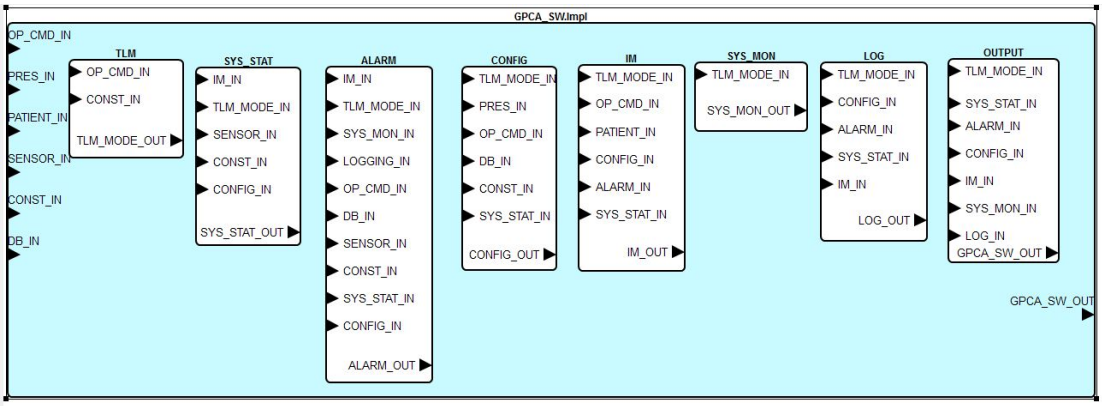


Figure 5.3: GPCA Software Architecture View

For instance, since our interest was exploring the GPCA software component, we defined the software architecture as show in Figure 5.3. We modeled the architecture in the Architecture Analysis and Description Language Notation (AADL) [76] using an open source tool called OSATE [72]. Our choice of this notation and tool combination is motivated by their capability to capture hierarchical architecture and rigorously perform

most of the analyses specified by our reference model.

AADL: It is a textual language that can be expressed graphically and is accompanied by a UML profile. AADL includes constructs that describe both software and hardware components, as well as mapping software components to physical resources and the devices with which they communicate. It allows for specification of interfaces for flow of control and data. The basic building block of this notation is a component, defined by its category (hardware, software, or composite), type (how the component interacts with the outside world), and its implementation (an instance of the component type). Note that there can be many instances of one component type. For example, highly available systems often have redundant computing resources to support fail-over; these can be represented as instances of a single component type. Our current GPCA example does not have redundant processing elements, but these may be added later. As as added benefit, AADL is supported by a growing number of tools, including tools that support editing and import/export of AADL models, as well as tools that allow one to analyze different aspects of the model—correctness of the connections, component resource usage within limits, etc.

While the choices we made for each component's interfaces may be subject to debate [31], in our work we do not attempt to suggest an optimal scope. We would like, however, to emphasize that whatever scope boundary is chosen, it must be well defined so that the requirements can be stated consistently. In fact, the scope chosen for the GPCA was influenced by the architecture of common medical devices.

5.2.2 Requirements Allocation and Adjustment

As we captured the architecture of the system, we specified (or allocated) requirements to each component in terms of the quantities of that component per our hierarchical

model. During this process, when we considered readily available components, such as a specific pump hardware, we had to adjust the quantities and requirements of its peer components in the decomposition to accommodate that pump component's quantities and requirements. Also, the limitation of the sensor—such as its tolerance—triggered us to alter the tolerance of the overall GPCA system. By using the Hierarchical reference model as a reference, we were able to traverse the "Twin Peaks" systematically.

The model also helped ensure that the requirements were allocated to the appropriate component, a major source of ambiguity and confusion in large, complex system requirements. For instance, one of the requirements that was inherited from an existing GPCA system-level requirements document specified that:

> *"When the option to suspend the pump is selected, the current pump stroke shall be completed prior to suspending the pump".*

When we decided to scope the GPCA system requirements at the level of nurse-patient-GPCA interaction, it became clear that this requirement does not belong at the system level, but rather concerns pump hardware. By systematically traversing through the architecture, we were able to identify that the requirement belongs to the pump hardware component. At the end of this exercise, the requirements of each component were expressed only in terms of the monitored and controlled quantities of each component, that is, the requirements were expressed at the appropriate level in the systems hierarchy and over a well defined component; this greatly improved the clarity of the requirements.

Further, as we defined the requirements, the properties specified in our reference model helped identify several issues. For example, the consistency property helped identify conflicts between two requirements of one of the software components (called the Infusion Manager):

The software shall not command any drug when there is an active alarm

The software shall command a flow at a certain rate when the system is in basal mode.

When we tried to establish the consistency property, we were able to identify that the requirements conflict when the alarm and the basal mode were simultaneously active. This helped us resolve the conflict by refining those requirements. While we used a formal tool to identify this conflict (that we elaborate in the next section), it can be identified through systematic, informal means as well.

5.3 Facets of Multi-level Requirements Flow-down

From the rough sketches of requirements to a properly expressed, consistent, relatively complete and well-structured requirements document, engineers take a long way to go in order to capture all relevant aspects or facets in the requirements. At each level of system abstraction, there are varying concerns that engineers focus while specifying the requirements. Such varying concerns induce the need to use a variety of models and requirement notations. In fact, such variations help effectively discover requirements.

In the following sections, we describe our experiences and challenges in progressively flowing down the requirements of the GPCA to each of its components in its architecture. While the challenges and approaches we took is specific to the GPCA, we believe that the experience in broadly applicable to the domain and our approaches will act as a catalyst for similar explorations.

5.3.1 Continuous Domain requirements

At higher levels of abstraction. Understanding the interaction patterns between the system and its environment as well as between the various hardware and software components within the system is crucial. The behavior of both the hardware components as well as the environment is continuous and continual in nature. Frequently, we found that the requirements for a system were focusing on the ordering of events in the system (as is done with various temporal logic) and the real-time constraints on these events. That information, however, is not sufficient to capture the continuous and continual nature of the system. For example, the rate of change of a controlled variable, the time it takes for a controlled variable to settle sufficiently close to a set-point, and the cumulative errors built up over time may be of critical importance, concepts that we found were not stated explicitly as requirements.

For example, let us consider a safety requirement of the GPCA,

> *"An over-infusion alarm shall be triggered if the flow rate of the drug in the infusion tubing is greater than X% of the prescribed flow rate for more than Y minutes"*

This requirement concerns the response to a flow rate of drug exceeding the prescribed value. Nevertheless, a closer examination of the problem reveals some ambiguity: Is it acceptable for the flow rate to exceed the threshold for Y consecutive minutes or does the time accumulate over the full infusion interval? Similarly, what is the cumulative effect of excess drug flow? For example, if the flow periodically exceeds $X\%$ for $Y - 1$ consecutive minutes and then comes back to normal for a short period of time, there is a possibility of over-infusion due the rate at which the drug is infused as well as the overall volume infused into the patient within a certain interval of time. We found behavioral modeling in the physical domain particularly useful in the elicitation, discovery, and refinement of such requirements.

Control System Modeling of GPCA

To understand the control behavior of the system in the continuous domain, we developed control system models – traditionally used to evaluate various control strategies, tune the controllers, etc.

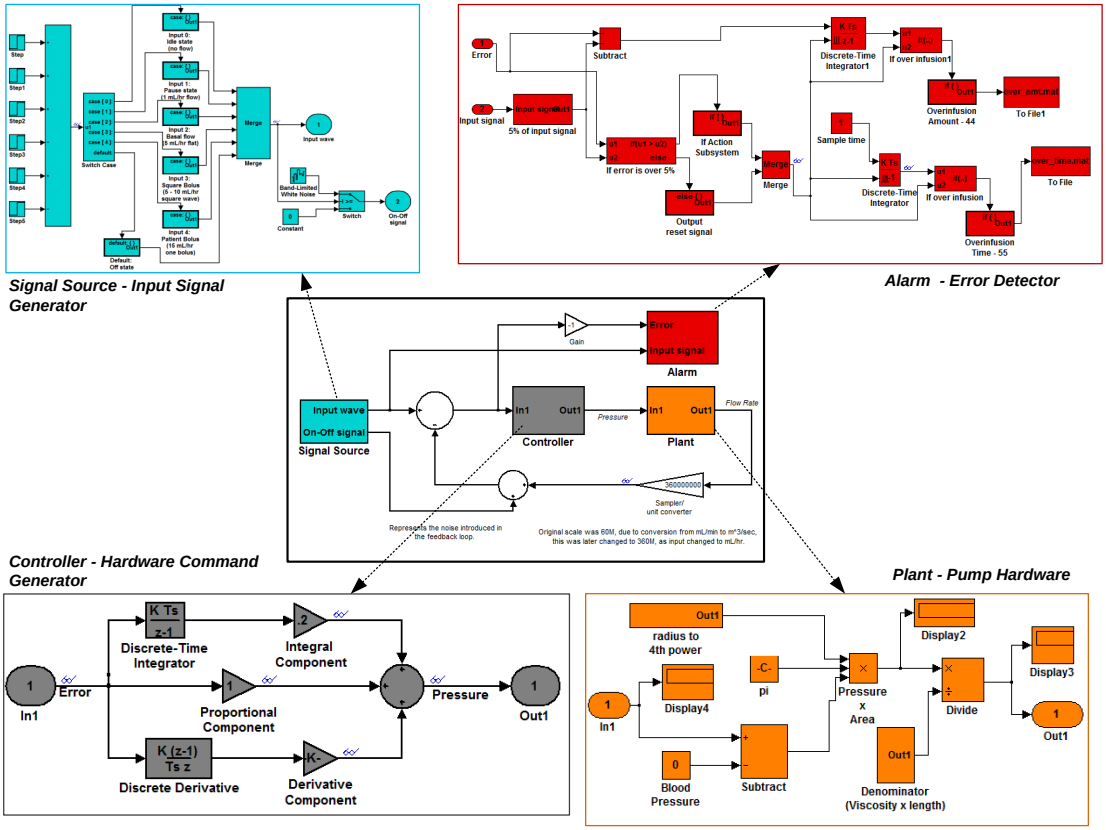


Figure 5.4: Control System Model

For the GPCA, we developed these models using MathWorks Simulink [60] tools to better understand the requirements needed to adequately constrain the desired system behavior (Figure 5.4) [39]. Through these modeling efforts we had an opportunity to explore various system responses, investigate how a proposed system might behave in its intended environment, and thereby identify precise requirements for the system. For example, the over-infusion safety requirement is now refined and stated as,

"The software shall issue an over-infusion alarm if the flow rate of the drug in the infusion tubing is greater than X% of the prescribed flow rate for more than Y consecutive minutes or more than Z minutes cumulative during the infusion duration."

"The software shall issue an over-infusion alarm if the volume of drug delivered in P consecutive minutes anytime during the infusion duration is more than Q% of the prescribed volume to be infused in that interval."

Through this modeling exercise, in addition to identifying precise requirements for the software, we were also able to identify several categories of requirements that should be considered while capturing requirements [39].

Classes of Requirements

As mentioned above, during our iterative requirements and modeling efforts working on the GPCA infusion pump, we realized that classes of requirements related to the continuous and continual behavior of the system were not captured in the available documentation. These missing or inadequate requirement generally fell into six categories closely related to the properties of the step response curves discussed in the previous section. The classification presented in this section is not a comprehensive list, rather to be viewed as a step towards capturing the classes of requirements that should be present in a requirements document or their absences should be justified.

To illustrate the requirements we will be primarily using the Actual Flow Rate of drug through the infusion hose in an infusion pumps system. Note that the requirements examples presented in this section are simply examples of what such requirements might look like; it is not intended to be complete and comprehensive in any way.

Accuracy

The accuracy (or precision) of the control of a physical quantity is quite obviously a concern and is typically defined in a requirements document. In our case, the accuracy of the flow-rate can be defined as a percentage of the flow-rate.

> *"The actual flow-rate (f) during normal operation shall be within ±5% of the target flow-rate (tfr): 0.95· tfr ≤ f ≤ 1.05· tfr."*

Naturally, the accuracy can be specified as an absolute value or as a value that decreases (as opposed to increases) with the flow-rate (the pump may be more accurate at higher flow-rates). Nevertheless, accuracy requirements for all controlled quantities are needed.

Rise Time (Drop Time)

The time allowed to go from one set-point—in our case typically basal flow-rate or a very low flow rate, such as Keep Vein Open (KVO) flow rate, to one of the bolus flow rates—cannot be more than a certain time interval. If the rise time is unconstrained, we may pick a solution that increases (or decreases) the quantity so slowly that it is harmful to the patient. In our case:

> *"The duration between the time at which a new target flow-rate (tfr) is commanded and the time at which the actual flow-rate (f) reaches within ±5% of the target flow rate shall be shall be less than 1.0 s."*

Requirements governing at least the maximum rise time (or drop time) are needed in all systems controlling physical quantities. A minimum rise time may be needed in case too rapid changes in a controlled quantity may be harmful. Such constraints on rise time could instead, however, be handled through requirements on the rate of change as opposed to rise time.

Rate of Change

Given a required rise-time, one can select different control approaches to reach the new set-point within the allocated time. For example, one can initially increase the flow rate rapidly and then taper off to smoothly capture the new target rate. Alternatively, one could rapidly increase the rate all the way to the new target rate and accept a larger overshoot (discussed next). Thus, in some instances (our automotive cruise control is a prime example), requirement on the rate of change may be needed—we may need to write requirement related to the first derivative of the controlled quantity.

> *"The rate of change in the actual flow-rate (f) shall not exceed 0.5 ml/s^2:*
> $\dot{f} \leq 0.5\ ml/s^2$."

In some systems we may even be interested in how quickly the rate of change changes. For instance, in our cruise control system, there will be requirements governing the rate of change in the speed (the acceleration of the vehicle—the first derivative of the controlled variable vehicle speed). In addition, we are most likely also interested in how quickly the acceleration changes (a concept known as *jerk*, the second derivative of the vehicle speed). Jerk is a crucial aspect of passenger comfort and constraints will need to be put on the acceptable jerk (think about how the negative acceleration when you brake for a red light suddenly goes away when the vehicle stops—you feel the "jerk").

Overshoot (Maximum Deviation)

The accuracy requirements on a system may be limited to its operation during normal operating conditions. During failure conditions, alarm conditions, and during transitions from one operational regime to another (often referred to as mode-changes), we may be willing to accept a larger deviation from the desired value—there may be an allowed maximum deviation (or maximum absolute quantity) that cannot be violated under any circumstance.

"The actual flow-rate (f) shall never exceed 10% of the target flow-rate (tfr): $f \leq tfr + 10\%$."

"The actual flow-rate (f) shall never exceed 10 ml/h: $f \leq 10$ ml/h."

Requirements in this category will both limit the allowable overshoot as well as put safe limits on the various physical quantities.

Settling Time

In the trade-off between rise time and overshoot, the system may experience some oscillation before the controlled quantities settle within the acceptable accuracy range. In our case, settling time turned out to be a non-issue (see Figure 4.1) since the system dynamics of the pump and fluid flow are not conducive to oscillation. If we want to constrain settling time, we need to fist define what "settling" means. In our case, we define selling as being within our normal operating accuracy, that is, within $\pm 5\%$ of the target flow-rate.

"The time between when a new target flow-rate (tfr) is commanded and the time the actual flow rate (f) settles shall be less than 1.2 s."

Determining whether or not requirements on settling time are needed must be determined through modeling or previous system experience. If these requirements are omitted, a clear justification for their omissions should be presented.

Cumulative Error

As discussed earlier, it is quite possible that there is a constant offset in the deviation between the actual values of a controlled variable and the desired values. Thus, errors

may accumulate over time even though the system looks well behaved at any instance in time. In the case of the GPCA infusion pump, the volume infused in the patient might—due to infusion inaccuracy—become unsafe. Therefore, requirements on the cumulative error (in our case, the volume infused into the patient over a specified period of time) may be needed. If we assume that the time interval of interest in the GPC infusion system is δ second, we could constrain the volume infused as follows:

> *"The actual volume infused over a time interval of δ cannot exceed the commanded volume to be infused by more than 0.1 ml:*
> $\int_{t}^{t+\delta} f\,dt \leq \int_{t}^{t+\delta} tfr\,dt + 0.1\ ml.$"

Note here that if we want to express this requirement formally over the controlled quantities, we will need to express it as an integral over the flow rate since we are not directly controlling the volume of drug infused. In our work with the GPCA infusion pump, there have been a number of—often conflicting and highly unclear—requirements putting limits on the flow rate as well as the volume to be infused.

We advocate to identify the monitored and controlled variables at the interface between the environment, we are attempting to control and the system, we are putting in place to do so and express the system requirements in terms of this interface [45, 74]. Note that discovering this interface is a another *Twin Peaks* activity related to the architecture of the system solution we put in place and has been discussed elsewhere [33, 7, 30, 31].

5.3.2 Mode Logic Requirements

At lower levels of system abstraction, the dynamic behaviors of systems are frequently defined in terms of operational *modes*, which are frequently viewed to be mutually exclusive sets of system behaviors [53]. The modes together with the rules defining the transitions between them are called *mode logic* [47]. Derivation of precise requirements

of the mode logic is challenging due to the plurality of modes and the complexity of the rules that govern the transitions.

In the GPCA, understanding the mode logic of the various infusion delivery types, such as basal or bolus, was nontrivial and we again relied on modeling to illuminate and resolve the issues.

Finite State Machine Models

To analyze the GPCA infusion mode logic requirements and behaviors, we modeled it as a finite state machine, using MathWorks Simulink and Stateflow [60]. A portion of the model of the GPCA infusion mode logic is shown in Figure 5.5. In our endeavor to model the mode logic [65], we identified requirements and modeling patterns and requirement scenarios not apparent before the modelling efforts.

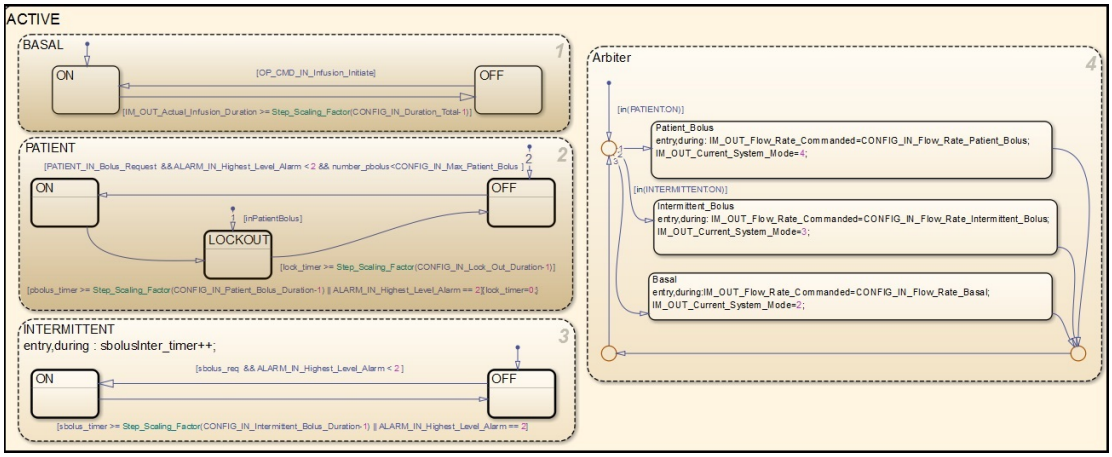


Figure 5.5: Infusion Mode Logic of GPCA

For example, a statement from one of the versions of the GPCA software's requirements document reads:

"A patient bolus shall take precedence over a square bolus. At the completion of the patient bolus, the square bolus shall continue delivery."

This requirement may be taken to mean that the patient bolus simply overrides the programmed square bolus infusion as illustrated in Figure 5.6(a) or the square bolus is suspended until the patient bolus is delivered and resumed thereafter as illustrated in Figure 5.6(b). Clearly, these alternatives influence how much drug a patient can receive over a bolus interval. Note here that both may be acceptable from a safety perspective; there may, however, be clinical differences making one approach more desirable than the other. The modeling efforts help identify and resolve such trade-offs.

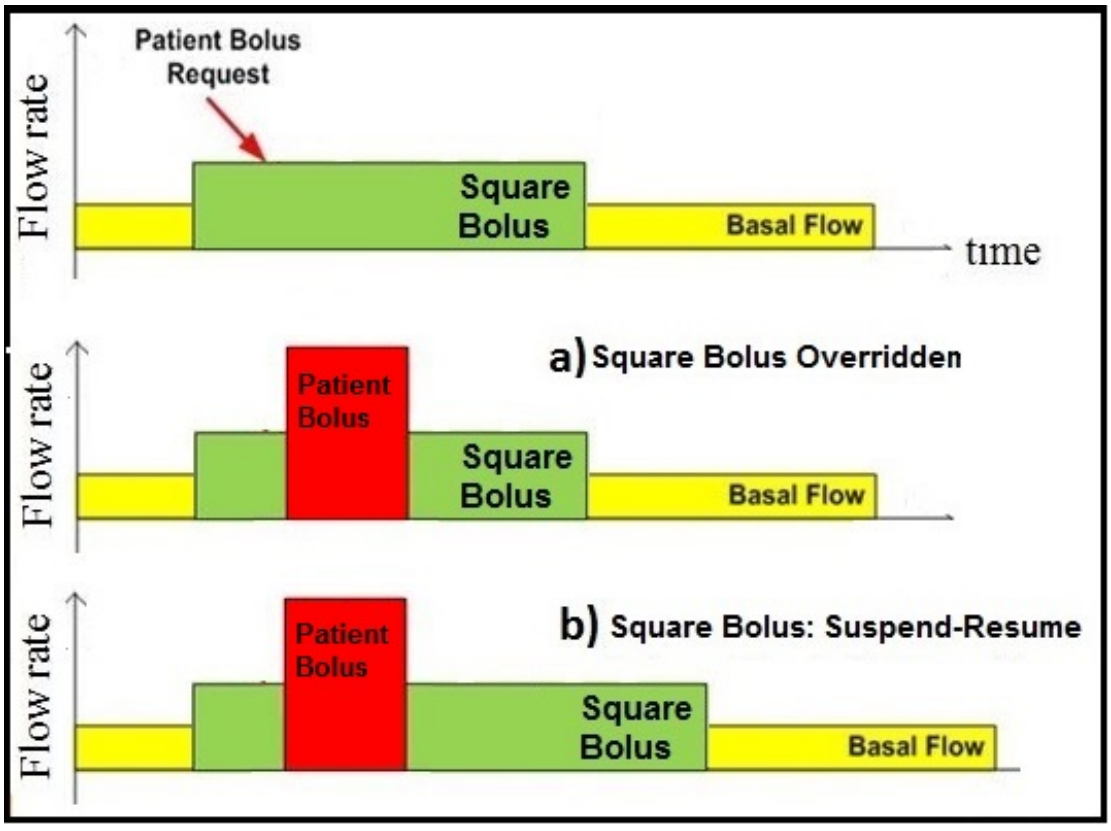


Figure 5.6: Mode Interaction Patterns

In addition, the modeling efforts helped identify the major operational modes and an overall conceptual structure of the GPCA behavior. This structure was later used to reorganize the natural language requirement to be conceptually cleaner and help understandability and readability. For example, in the GPCA natural language requirements document, we organized the requirements in a hierarchical structure reflecting the model structure of the system; the requirements of the parent level modes are also applicable to all the child modes. For example, all requirements applicable for the Active mode are also applicable in the Basal mode (sub-mode of Active). This organization enhanced clarity and reduced the repetition of a requirement in multiple places within the same document.

5.4 Summary

Summarizing the hierarchical requirements exploration and specification effort, our reference model provided the clarity to approach requirements and architecture in a organized and structured manner. While the model provided high-level guidance to specifying requirements, it does not prescribe to a specific approach, notation or expressibilty of requirements at any level of detail of the system. This is intentional, rather advantageous, since depending on the issues to be analyzed in each domain, the choice of the appropriate notation or level of detail can be chosen by the engineer. In the work described in this chapter, we demonstrated how the reference model provides abstract guidance, yet leaves adequate room for engineers to explore and define requirements from any perspective or level of detail. However, based on our experience, we find that the use models to symbiotically explore requirements and architecture was practically useful.

Chapter 6

Hierarchical Requirements Satisfaction

When hierarchically capturing the requirements and architecture of complex systems, ensuring whether their are indeed adequate and correct at each level in the hierarchy as well as at the overall system level is inevitable. The various properties defined in our reference model (Chapter 3) provide a rigorous mathematical foundation to perform such analysis. To practically establish those properties on real systems, mathematical techniques such as formal methods provide frameworks within which one can specify and verify systems in a systematic manner. In fact, to verify the requirements that we have captured for the GPCA, that we described in the previous chapter, formal methods provides a suitable means. However, when establishing those properties for large-scale systems in practice, we found that one has to cope with engineering challenges.

In this chapter, we discuss some of the challenges that we encountered when using mathematical techniques such as formal-methods that have been proven to be highly rigorous the level of rigor in verifying the requirements. A fundamental challenge with most formal approaches is that do not naturally support verification of the hierarchically

intertwined requirements and architecture; rather, they conceptually treat the verification of requirements and architectures as separate from each other. Hence, an approach that supports verification of the intertwined architecture and requirements in a scalable manner is desirable.

Further, as described in the previous chapter, as one organizes requirements and models of complex systems into multiple abstraction layers, where each layer focus on a different concerns, the analysis domain has to be suitable for that level of abstraction. For instance, the analysis at higher-levels have to focus on the requirements concerning the interactions of the system with its physical environment in non-deterministic and continuous time-domain, whereas at lower-levels such as software the requirements focus on discrete, deterministic behaviours. Hence, we need to resot to a multi-domain compositional analysis paradigm, where care must be taken to ensure that the levels of abstraction are seamlessly and rigorously glued, results from one analysis tool can be used in another to avoid errors due to "mismatches" in the semantics of the underlying formalisms.

In the rest of this chapter, we describe an end-to-end scalable approach to establish the properties (defined in the reference model) of the hierarchically intertwined requirements and architecture of complex systems. We use the closed loop clinical scenario of GPCA to illustrate our approach. First, we devise strategies to independently verify the properties of each layer of system abstraction using tools and techniques appropriate for that layer. Next, we demonstrate an approach to soundly combine the analysis between the layers. We highlight the key aspects of our approach for providing a semi-formal semantic mapping to tie distinct verification paradigms and for reconciling the abstraction induced differences between the closed-loop system and the infusion pump system. While the specifics are particular to the case example at hand, we believe the approach is broadly applicable in similar situations for verifying complex system properties.

6.1 Verification Approach Overview

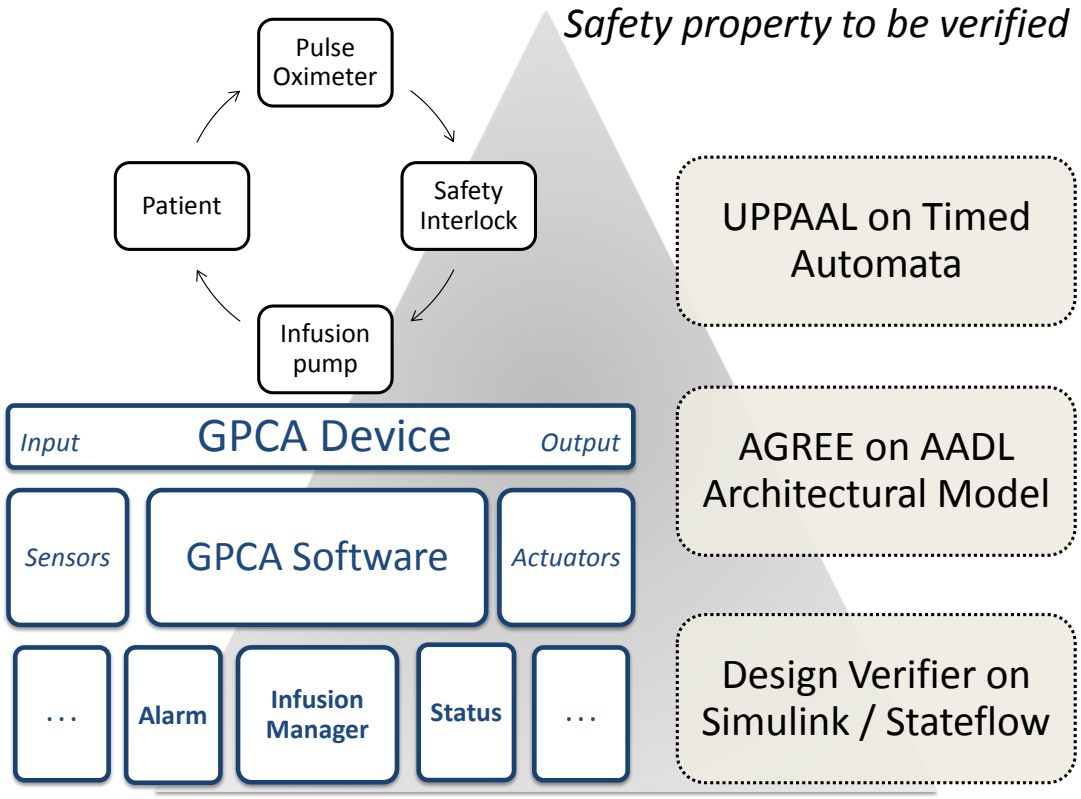


Figure 6.1: Multi-model verification

Our strategy for verifying properties of systems, such as the closed-loop infusion system whose artifacts span multiple abstraction levels is depicted in Figure 6.1. The main idea is to use the notation and tools appropriate for each abstraction level, decompose the single verification problem into distinct verification tasks for each level of abstraction, and finally tie together the results in a logical fashion that is amenable to manual review.

For the stratified verification of the GPCA, we considered three different sets of notations, tools, and techniques. At the highest level of system abstraction, that was

modeled using the timed automata, we used a tool named UPPAAL that helps establish the requirements of the overall system in the hospital environment. Next, to verify the hierarchical architectural decomposition of the system into its components and the components into its sub-components, we used the recently developed JKind tool[1] by Collins Aerospace. At the lowest-level, where sub-components behaviours were modeled in detail, we used the Simulink Design Verifier (SLDV) [59] to ensure if the modeled behaviours indeed uphold the sub-component's requirements. To logically glue the independently verified levels of abstraction such that one can reason about the end-to-end properties of the system, we use a combination of formal and semi-formal approaches. In the rest of the chapter, we elaborate these approaches.

6.2 "Twin Peaks" Verification

To establish the hierarchical requirements satisfaction argument (defined in 3.22), one needs to simultaneously consider the requirements between adjacent layers of system abstraction and the architecture that relates them. In the previous chapter, we described how we used AADL to describe the architecture and AGREE notation to express assume-guarantee contracts (aka requirements) at the software level of the GPCA. To verify if the specification and model of the GPCA are correct, we use AGREE [13] – a compositional reasoning framework based on assume-guarantee contracts. AGREE is a plugin to the OSATE AADL tool [72] and adds support for requirements capture and formal verification of the architectural models.

6.2.1 AGREE

AGREE (Assume Guarantee Reasoning Environment) is a compositional reasoning framework based on assume-guarantee reasoning developed by Collins Aerospace and

[1] Available at: https://github.com/agacek/jkind

University of Minnesota. AGREE was primarily developed to validate system design by reasoning about the flow down of requirements with respect to the architecture. The framework is used to prove that the system requirements are established, given the architectural structure of the system and the requirements allocated to sub-systems in the architecture.

To understand the verification performed by AGREE, consider a toy example with a simple architecture as shown in Figure 6.2. In this example, we would like to establish at the system (S) level that the output signal is always less than 50, given that the input signal is less than 10. We can prove this using the assumptions and guarantees provided by the sub-components A, B, and C that are organized hierarchically. This figure shows one layer of decomposition, but the idea generalizes to arbitrarily many layers. We want to be able to compose proofs starting from the leaf components (those whose implementation is specified outside of the architecture model) recursively through all the layers of the architecture.

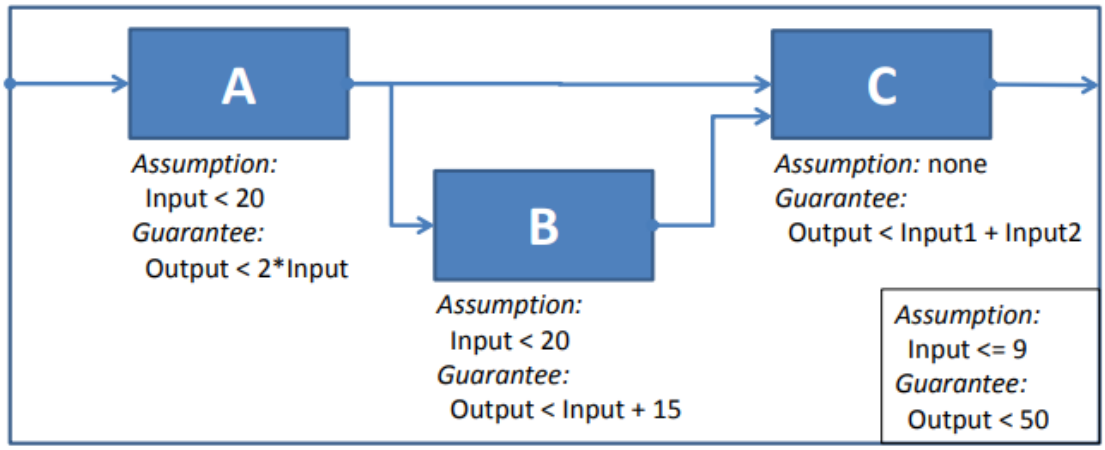


Figure 6.2: Toy Architecture with Properties

The correctness obligations established in AGREE tool are the form G(H(A) $\Longrightarrow$

P), which informally means that it is always the case that if assumption A has been true from the beginning of the execution up until this instant (A is historically true), then guarantee P is true. For the obligation in Figure 4, the goal is to prove the formula G(H(AS) $\implies$ PS) given the contracted behavior G(H(Ac) $\implies$ Pc) for each component c within the system. To prove the obligation, we establish generic verification conditions that together are sufficient to establish the goal formula. In the example, this means that for the system S we want to prove that Output < 50 assuming that Input < 10 and the contracts for components A, B, and C are satisfied. For a system with n components there are $n + 1$ verification conditions: one for each component and one for the system as a whole. The component verification conditions establish that the assumptions of each component are implied by the system level assumptions and the properties of its sibling components. For this system the verification conditions generated would be:

$G(H(AS) \implies AA)$

$G(H(AS \wedge PA) \implies AB)$

$G(H(AS \wedge PA \wedge PB) \implies AC)$

$G(H(AS \wedge PA \wedge PB \wedge PC) \implies PS)$

In general, these verification conditions may be cyclic, but if there is a delay element in the cycle we can use induction over time as in [27]. The system level verification condition shows that the system guarantees follow from the system assumptions and the properties of each sub-component. This is essentially an expansion of the original goal, G(H(AS) $\implies$ PS), with the additional information obtained from each component.

6.2.2 AGREE and The Hierarchical Reference Model

Although the reference model and AGREE were developed independently for different purposes, AGREE can be envisioned as an exemplar of using the Hierarchical reference

for synchronous discrete systems analysis. AGREE's notion of hierarchical requirement refinement along the architectural lines supports the Hierarchical reference model. Further, the architectural constraints (3.18, 3.19) and requirements satisfaction (3.22) specified in the Hierarchical reference model is the underlying principle for compositional verification in AADL and AGREE. AGREE is used to verify, if for a given layer of the architecture, the contracts of the sub-components within the architecture is sufficient to establish the system level goals of that layer. This exactly reflects the hierarchical requirements satisfaction in the reference model.

However, the Hierarchical reference model is much more generic than AGREE. AGREE was primarily designed and developed for reasoning about synchronous discrete control systems, the notations for modelling and expressing requirements of the system have limitations, that are acceptable for the types of systems AGREE was intended analyze. For example, properties such as unbound liveliness of systems are not verifiable using AGREE; but AGREE can verify bounded liveness properties (bounded over time) that are acceptable for critical systems. Similarly, the Hierarchical reference model specifies the requirements to be expressed as artifact R, but in AGREE these are specified as contracts (Assume-Guarantee pairs). Again, for the type of systems analyzed by AGREE, this representation is adequate.

In sum, although AGREE was developed in an independent setting without referring to the Hierarchical reference model, it can be envisioned as an exemplification of the reference model simplified for the analysis of the synchronous discrete control systems. In the next section, we illustrate the GCPA system and software's hierarchical, "Twin Peaks" verification using AGREE.

6.2.3 GPCA Verification in AGREE

In order to illustrate AGREE's verification of a GPCA software, consider one of its system requirement:

> *When the system was in the infusing mode and if the estimated drug remaining in the drug reservoir drops below the empty drug threshold (estimated-drug-remaining < empty-drug-threshold), the GPCA software shall issue visual-alarm and aural-alarm commands, and stop the infusion.*

Based on the high level functions of the components, four out of the eight GPCA software components were relevant for this requirement, namely TLM (Top Level Mode), SYS_STAT (System Status), ALARM (Alarm) and IM (Infusion Manager). Each of these components have their own requirement that has been flown down to them from their parent system level. The requirement of **TLM** component - responsible for signaling whether the system is in an operable condition (ON) - is :

> *The TLM subsystem shall always issue ON whenever a START command is received.*

The **SYS_STAT** component, is responsible for identifying the system state and computing remaining reservoir volume based on the feedback from the other components. Hence the requirements flown down to this component are:

> *While ON, the SYS_STAT subsystem shall always output infusing=true if the system mode input > 1.*

> *While ON, the SYS_STAT subsystem shall always output empty_reservoir = true if the estimated drug remaining in the drug reservoir < the empty drug threshold*

Similarly, the **ALARM** component, that monitors exceptional conditions, is allocated with the requirement:

> *While ON, mode* > 1 *and empty reservoir = true, the Alarms subsystem shall output the alarm level to 4 (critical alarm).*

Finally, the **Infusion Manager(IM)** component - responsible for commanding the flow rate - is allocated with a requirement:

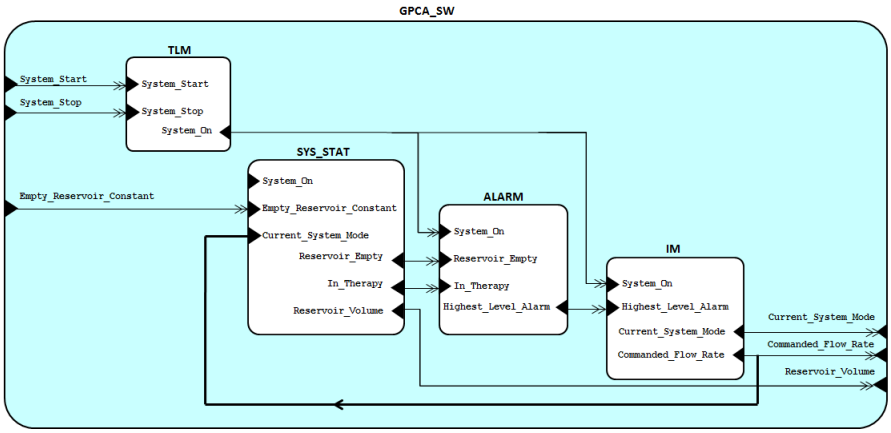
> *When ON, the infusion manager subsystem shall stop infusion whenever a critical alarm occurs (alarm level=4)*

Using these component's contract as guarantees, the system level requirement is verified using AGREE. A snapshot of the system model and the formalized representation of the requirements are shown in Figure 6.3. The verification conditions of AGREE, shown at the end of the Figure 6.3, is comparable to *Requirements Satisfaction* property defined in the reference model.

6.3 Behavioural Verification of Leaf-level Components

As mentioned in the previous section, the leaf-level component's behaviors of the GPCA were modeled using MathWorks Simulink and Stateflow tools. While there are several tools to verify if the behavioral models satisfy their requirements, we choose Simulink Design Verifier (SDV) [60], a plug-in tool for formal verification developed by MathWorks.

Ideally, the component requirements captured in AGREE would be reused for verification of the component behaviors. Unfortunately, the verification tools available for Simulink and Stateflow do not currently support AGREE's formalism for property specification. Naturally, it would be highly desirable if the property specification language



GPCA Software

m_0^0 : System_Start, System_Stop, Empty_Reservoir_Constant
c_0^0 : Current_System_Mode, Reservoir_Volume,Commanded_Flow_Rate
R_0^0 : property emp_res_no_flow = true -> (System_Start and not(System_Stop) and (pre(Current_System_Mode) > 1) and (Reservoir_Volume < Empty_Reservoir_Constant))=> Commanded_Flow_Rate = 0);

TLM Component

m_1^1 : System_Start, System_Stop
c_1^1 : System_On
R_1^1 : property sys_start = (System_Start and not(System_Stop) => System_On)

SYS_STAT Component

m_2^1 : System_On, Empty_Reservoir_Constant, Current_System_Mode
c_2^1 : Reservoir_Volume, Reservoir_Empty, In_Therapy
R_2^1 : property emp_res = (System_On and (Reservoir_Volume < Empty_Reservoir_Constant)) => Reservoir_Empty;

ALARM Component

m_3^1 : System_On, Reservoir_Empty, In_Therapy
c_3^1 : Highest_Level_Alarm
R_3^1 : property alm_lvl = (Highest_Level_Alarm >=0 and Highest_Level_Alarm <= 4);
property emp_res_alm_L4 = System_On and In_Therapy and Reservoir_Empty => Highest_Level_Alarm = 4;

IM Component

m_4^1 : System_On, Highest_Level_Alarm
c_4^1 : Commanded_Flow_Rate, Current_System_Mode
R_4^1 : property alm4_no_flow = ((System_On and Highest_Level_Alarm = 4) => (Commanded_Flow_Rate = 0));

Requirements Satisfaction : $(R_1^1 \wedge R_2^1 \wedge R_3^1 \wedge R_4^1) \Rightarrow R^0$

Figure 6.3: AGREE Reasoning of empty reservoir property

Figure 6.4: Simulink verification block with Embedded MATLAB Code

was consistent throughout the modeling effort; this is a tool integration and engineering problem that we are exploring independently. For this case study, we recaptured the required component properties for verification in one of the notation supported by the Simulink Design Verifier.

The SDV requires the properties to be specified as Boolean expressions in one of the available MATLAB notations. One of the ways, as depicted in Figure 6.4, is capturing it as a Simulink verification block. The input signals on the left side are the same inputs that are provided to the Alarms component capturing the component behavior. The gray circle containing a P (for property) indicates that the verification tools will attempt to verify that this Simulink block always generates a signal that is True; if the signal is ever False, the tools have revealed a property violation and will report a counterexample. The logic of the verification condition is in Figure 6.4 expressed using embedded MATLAB, a subset of the MATLAB computing language that supports efficient code generation for deployment in embedded systems. Alternatively, the verification conditions could be captured using the Simulink or Stateflow notations. The same condition

captured as a Simulink model can be seen in Figure 6.5.

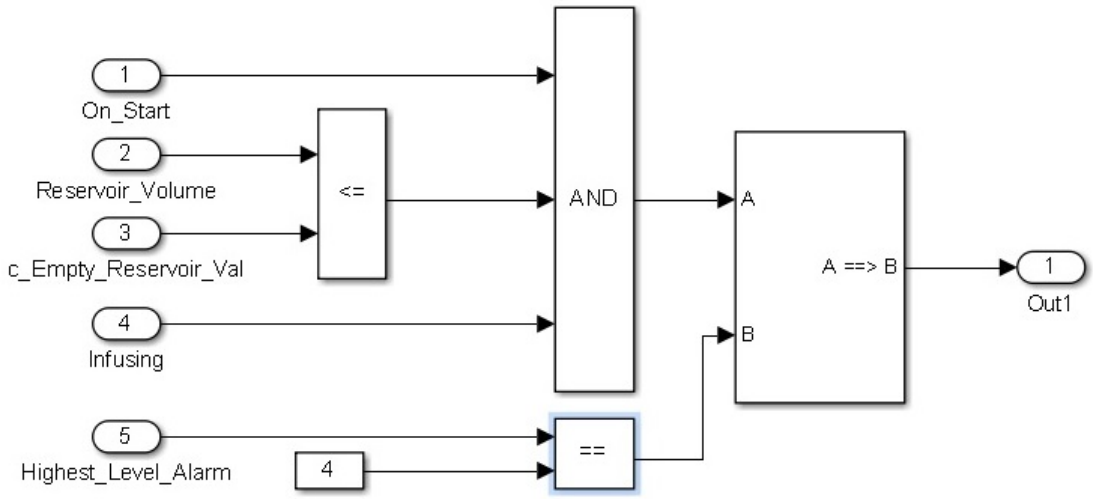


Figure 6.5: Property expressed as Simulink Model for Verification

For our work, we captured the properties using embedded MATLAB notation for several reasons. First, the property can be structured to closely resemble the natural language "shall" requirements in the requirement document as well as the PLTL properties used in AGREE. This closeness in structure reduces the opportunities for transcription mistakes and makes the properties easier to inspect. For example, the correspondence between the properties are illustrated in Figure 6.6. In the future, the translation between the AGREE properties and the Design Verifier properties will be automated. Second, the textual notations makes it easy to comment out properties as the verification process is underway (verifying all properties all the time may be a waste of time when there is one problematic property of interest). Finally, we have developed preference for the textual notation since we find it quicker and easier to create and maintain the required properties textually.

```
property no_enter_therapy =
        true ->
           pre(IM_OUT.Current_System_Mode) = 1 and not(OP_CMD_IN.Infusion_Initiate and CONFIG_IN.Configured > 0)
               => (IM_OUT.Current_System_Mode = 1);
guarantee "no_enter_therapy":no_enter_therapy;
```

Property of Infusion Manager Component in AGREE

```
sldv.prove(no_enter_therapy);
function result = no_enter_therapy(pre_Current_System_Mode, Infusion_Initiate, Configured, Current_System_Mode)
     checkCondition = (pre_Current_System_Mode == 1 && not(Infusion_Initiate && Configured > 0));
     checkOutput = (Current_System_Mode == 1);
     result = implies(checkCondition, checkOutput);
```

Property of Infusion Manager Component in Embedded Matlab

Figure 6.6: Property correspondence between Embedded MATLAB and AGREE

6.4 Real Time-Domain Verification

At the highest level of abstraction, we had modeled the closed-loop clinical scenario for GPCA using the Timed Automata notation. This prototypical model included a patient blood oxygen level monitor whose output is constantly monitored by a supervisory controller that can automatically command the pump to terminate (pain medication) infusion, should the oxygen-level fall below a critical threshold. While the scope of the verification of the closed loop system is not in scope of our work, in this section, we focus on discussing the compositional approach, hierarchical approach to verify GPCA system level properties in the closed loop system.

We consider two essential closed-loop safety properties. The first one states that, once the patient's blood oxygen saturation level becomes lower than 90%, the pump will eventually be stopped, expressed as an UPPAAL query as `samplebuffer < 90 --> PCAbasal.stopped`. Here, `samplebuffer` is a shared variable that represents the true oxygen saturation. Note that the invariant of the `stopped` state in the `PCAbasal` automaton is `pca_rate==0`, implying that no drug is entering the patient. The second property takes into account the fact that the stopped pump can be restarted manually

by the caregiver. We show that if the caregiver never restarts the pump when the patient is in danger, the patient eventually recovers, with the oxygen saturation levels returning to normal. This property is expressed as an UPPAAL query as `samplebuffer < 90 --> samplebuffer >= 91`.

6.5 Mapping Verification Paradigms

While these closed properties were proved using a tool called UPPAAL [73] and GPCA system level properties were verified using AGREE tool, a natural question then is whether these two analyses can be combined in some meaningful way such that the particular infusion pump when used as part of the closed-loop system can be guaranteed to uphold critical safety properties. In the rest of this section we demonstrate an approach to soundly combine these analysis results. We highlight the key aspects of our approach for providing a semi-formal semantic mapping to tie distinct verification paradigms and for reconciling the abstraction induced differences between the closed-loop system and the infusion pump system. While the specifics are particular to the case example at hand, we believe the approach is broadly applicable in similar situations for verifying complex system properties.

6.5.1 Matching Abstractions

To ensure that the concrete GPCA works correctly in the closed-loop instance, it is necessary to show that the infusion pump in the closed loop model *simulates* the concrete GPCA. We are currently extending AGREE tools to support a machine-checked proof of simulation. In this section, we discuss how we relate the abstract and concrete models and our (currently informal) check that the relation is a simulation.

Defining Mappings

To create a simulation relation, one must match the variables and states in the abstract model with those in the concrete model. The concrete GPCA model has many inputs, states and transitions that are not represented in the abstract model. We addressed this by associating concrete infusion modes with abstract states of the infusion pump, introducing a non-deterministic input in the abstract model, and finally adding environmental assumptions in the concrete model to prevent it from entering states that do no preserve the mapping.

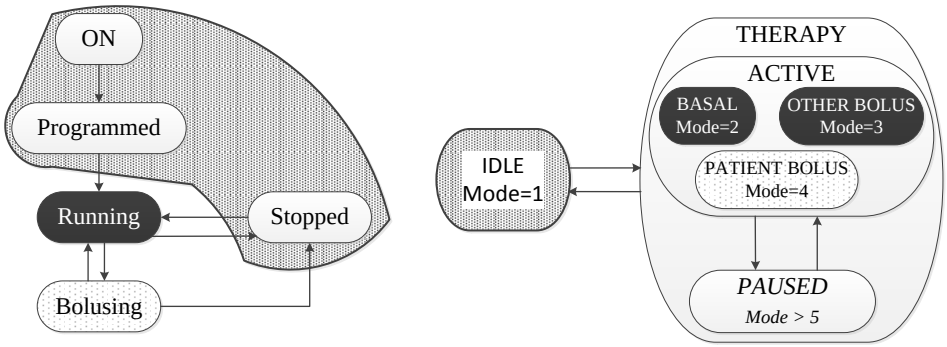


Figure 6.7: Abstract PCA and Concrete GPCA High level system state mapping

First we established a mapping between states in the abstract model of the infusion pump and the infusion modes of the concrete GPCA model, as shown in Figure 6.7. This enables restating properties specified in terms of flow-rate constraints at the abstract level as properties in terms of the infusion modes. The different GPCA infusion modes have specific associated flow-rates that can be immediately verified for conformance to flow-rate constraints. In this mapping, the abstract states `Stopped`, `ON` and `Programmed` are associated with the `IDLE` mode represented by the predicate `Current_System_Mode` $= 1$; `Running` is associated with `BASAL` and `OTHER BOLUS` modes represented by `Current_System_Mode` $\in \{2, 3\}$; and `BOLUSING` is associated with

PATIENT BOLUS represented by Current_ System_Mode = 4. Note that there is no abstract state associated with the PAUSED mode of the GPCA, which denotes a temporary halt to infusion due to certain exceptional conditions identified by the alarm sub-system. Environmental assumptions, to be discussed shortly, guarantee that this mode is not reached for the scenarios considered.

Second, the abstract closed-loop system modeled the "stop" command to the pump as an exclusive input from the supervisory control for oxygenation-level based safety-interlock. However, realistically the operator may command the GPCA to stop at any time – one may view the operator as an external monitor for exceptional conditions, analogous to the internal monitor represented by the ALARMS component. Unlike infusion pump alarms, which are excluded from consideration at the abstract level, the "stop" command was included in the original abstract model, albeit for a restricted purpose. By allowing the "stop" to occur non-deterministically, one can easily simulate an externally commanded infusion stoppage. The Pump_Stop input signal was made non-deterministic in the abstract model to effect this change.

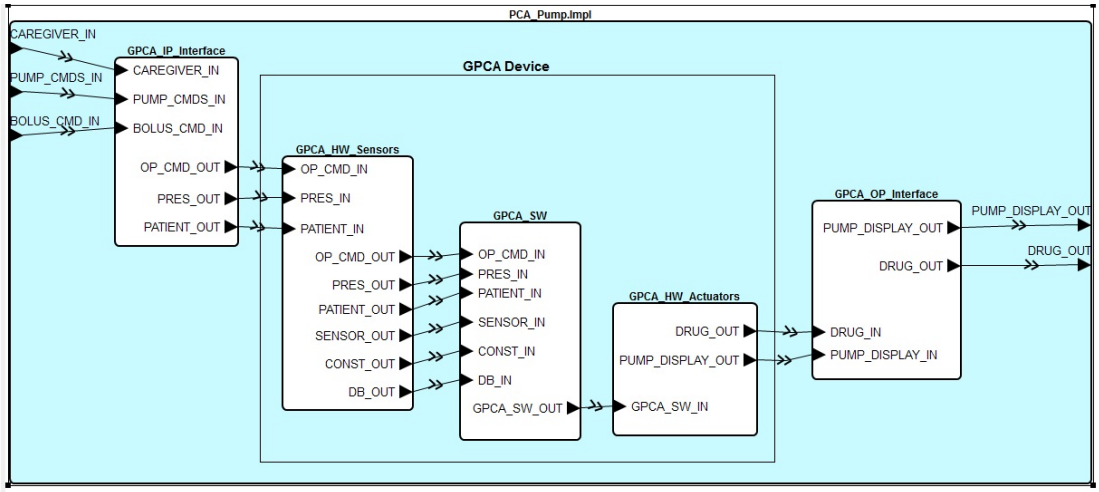


Figure 6.8: GPCA Architecture with PCA Interfaces

Finally, to match the inputs and outputs of the two modelling paradigms, we refined the architectural model of the GPCA at the system level. In the architectural model of the top most system, `PCA_Pump`, we included interfaces (inputs and outputs) that match the PCA interfaces of the closed loop system modeled in UPPAAL. This was done so that the level of abstraction of the `PCA_Pump` in AADL is same as the PCA of the closed loop system in UPPAAL and the properties of the `PCA_Pump` in AADL are always specified at the same level of detail as in the UPPAAL model.

The concrete model has more inputs than the abstract `PCA_Pump` model; hence the interface component matches some straightforward inputs, such as `Pump_Stop` of the abstract model to `Infusion_Cancel` of the concrete model, as well as suppress certain concrete model inputs so that the behaviours of the concrete model not required by the abstract model are excluded. For example, the concrete model implements a functionality of suspending infusion in response to an `Infusion_Pause` command from the clinician. However the abstract model does not specify this functionality. Hence the interface suppresses the `Infusion_Pause` command to the device. Similarly other inputs and system conditions of the concrete model that are not specified by the abstract model are abstracted by the interfaces.

6.5.2 Aligning Hierarchical Models and Properties

In order to formally link the closed loop PCA to the architectural model, we recaptured the closed loop PCA properties as AGREE contracts at the `PCA_Pump` level in AADL. Let us take an example to explain the hierarchical steps of verification. A closed loop level property for the PCA stated informally is, *Infusion cannot begin until parameters of the infusion are configured and pump is started by the caregiver.* This requirement recaptured as AGREE contract is,

```
property no_infusion_start  =
```

```
true -> pre(PUMP_DISPLAY_OUT.Current_System_Mode)=1
  and (not(CAREGIVER_IN.Infusion_Start
    and CAREGIVER_IN.Infusion_Programmed)) =>
      (PUMP_DISPLAY_OUT.Current_System_Mode = 1);
```

In the above formulation, the `pre` of the variable `(Current_System_Mode)` represents its value from the previous step. The `(Current_System_Mode = 1)` represents the pump state (abstractly represented as mode 1) in which flow rate is zero. The astute reader might have noticed the ``‘‘true’’`` at the beginning of the property, which effectively makes the property true in the first step. This is intentional, for otherwise the previous value of the variable at the very first step could be an arbitrary value that causes some avoidable complications. To keep the specifications straightforward, we defined properties such that, we specify that the pump always starts in a state at which flow is zero and then onwards the above mentioned property holds.

In order for the above property of `PCA_Pump` to hold, its components should compositionally guarantee using their properties. The `GPCA_SW` being a component of `PCA_Pump` has much richer functionality than that specified by `PCA_Pump`. Hence state mapping between them was required to write `GPCA_SW` properties that would guarantee `PCA_Pump` properties. Figure 6.7 shows a high level representation of the states and its mapping between the systems. The state representation of `PCA_Pump` was derived from the UPPAAL model and that of `GPCA_SW` was derived from its behavioral specifications. Similar to the state mapping, the transitions from and to these states were also mapped. We discuss the details in a subsequent section.

In order to satisfy the `PCA_Pump` property, certain assumptions were required at the `PCA_Pump` level to match the abstractions. First of all, `CAREGIVER_IN.Infusion_Programmed` - specifies if a valid prescription is already programmed in the pump - was mapped to the concrete model variable `(PUMP_DISPLAY_OUT.Configured` > 0`)` that

means the device holds a valid prescription. At the PCA_Pump level, these were still abstract concepts since the caregiver decides if the infusion is programmed by looking at the pump display and determining if it is configured. Hence we state this as an assumption at the PCA_Pump level.

Secondly, we mapped the (ON->Programmed) and Stopped of the abstract model to IDLE state of the concrete model hence the transitions (ON->Programmed)->Running triggered by (CAREGIVER_IN.Infusion_Start and CAREGIVER_IN.Infusion_Programmed) and the transition Stopped -> Running triggered by (CAREGIVER_IN. Clear_Command) were assumed to imply each other.

Thirdly, GPCA_IP_Interface maps the abstract model's Infusion_Start input signal with the concrete model's Infusion_Initiate command. Finally the GPCA_SW guarantees that,

```
property no_enter_therapy =
  true ->  pre(GPCA_SW_OUT.Current_System_Mode) = 1
      and not(OP_CMD_IN.Infusion_Initiate
        and GPCA_SW_OUT.Configured > 0) =>
          (GPCA_SW_OUT.Current_System_Mode = 1);
```

There were a few other component guarantees that deal with mapping inputs and outputs that are not discussed here since its not a significant point of discussion. Hence when we compositionally verified the PCA_Pump property in AGREE, it was guaranteed using their component guarantees and system assumptions.

At this point, the satisfaction of the properties were only until the GPCA Device components level. As discussed in Chapter 5, the GPCA_SW was further decomposed into components. Hence the properties guaranteed by the GPCA_SW for the PCA_Pump has to be guaranteed by its sub-components. We performed another level of AGREE reasoning in which now the GPCA_SW is the top level system that has properties to

be guaranteed (similar to the `PCA_Pump`) by its components guarantees such as `TLM, ALARM, IM`. For example, the property of the `GPCA_SW` (`property no_enter_therapy`)), is guaranteed by Infusion Manager(`IM`), Configuration Manager (`CONFIG`), and Output (`OUTPUT`) component guarantees. There were not as many assumptions and abstraction mapping required for the property guarantee between the `GPCA_SW` and its components since they were already done as a part of another initiative [67].

Establishing Simulation

The GPCA monitors various exceptional conditions as modeled by the `ALARMS` component and responds appropriately, some of which would include a temporary pause to infusion. These alarms may be triggered by sensor inputs such as *air-in-line* or internally computed conditions such as *empty-reservoir*. Thus, even a simple abstract property, *Once the pump is started, unless it is stopped, the drug flows at least at basal flow rate*, does not hold for the concrete model, because, the property presupposes the absence of infusion-ending triggers other than the stop command. In the closed-loop system, the abstract infusion pump model rightly disregards such details; at that level, one is interested in establishing essential system properties using a simplified notion of an infusion pump. The responsibility for stopping infusion for any reason can be pushed out to the environment – in this case the pump operator can be assumed to command "stop" as necessary. Therefore, when relating abstract infusion pump properties to a concrete realization of the pump, it is reasonable to interpret those as being prefixed with a standard caveat: "Absent exceptional pump conditions that internally trigger stoppage of infusion ...". In other words, we approximate each internal system failure as an external command to stop the pump. Figure 6.9 shows a snapshot of assumptions and guarantees that formally establishes the described abstraction mapping. Of course, these exceptional conditions could vary based on the specific pump under consideration.

In the process of mapping the abstract infusion pump model and its properties to the concrete GPCA, absence of these internally triggered exceptional conditions is captured as a system assumption: `no_cancel_implies_no_stop_conditions`. Such assumptions must then be discharged using properties of the environment in which the GPCA operates. In the present case, one has to manually review these assumptions to ascertain that those are acceptable caveats to the property of interest. This is a desirable outcome, since it calls attention to those exceptional conditions for the specific GPCA device.

```
-- UPPAAL Property :Pump will deliver at basal rate if the pump is not stopped.
property infusion_continue  =
 true -> (pre(PUMP_DISPLAY_OUT.Current_System_Mode) >= 2 and
    pre(PUMP_DISPLAY_OUT.Current_System_Mode) <= 4) and
    not(PUMP_CMDS_IN.Pump_Stop) => (PUMP_DISPLAY_OUT.Current_System_Mode >= 2 and
        PUMP_DISPLAY_OUT.Current_System_Mode <= 4);
guarantee "infusion_continue":infusion_continue;

-- Map modes with flow rates.
guarantee "Mode 1":(PUMP_DISPLAY_OUT.Current_System_Mode = 1) =>
                        (DRUG_OUT.Drug_Flow_Rate = 0);
guarantee "Mode 2":(PUMP_DISPLAY_OUT.Current_System_Mode = 2) =>
                        (DRUG_OUT.Drug_Flow_Rate >= CAREGIVER_IN.Normal_Infusion_Rate);
guarantee "Mode 3":(PUMP_DISPLAY_OUT.Current_System_Mode = 3) =>
                        (DRUG_OUT.Drug_Flow_Rate >= CAREGIVER_IN.Normal_Infusion_Rate);
guarantee "Mode 4":(PUMP_DISPLAY_OUT.Current_System_Mode = 4) =>
                        (DRUG_OUT.Drug_Flow_Rate = CAREGIVER_IN.Bolus_Infusion_Rate);
....
-- Maps pump stop input of abstract model to Infusion cancel of concrete model
guarantee "Pump Stop means Infusion cancel" :
        (PUMP_CMDS_IN.Pump_Stop <=> OP_CMD_OUT.Infusion_Cancel);
....
-- sensor conditions that cause infusion to be stopped.
eq sensor_conditions_that_cause_pause:bool =
    (SENSOR_OUT.Battery_Depleted or SENSOR_OUT.RTC_In_Error or
        SENSOR_OUT.CPU_In_Error or SENSOR_OUT.Memory_Corrupted or
        SENSOR_OUT.Pump_Too_Hot or SENSOR_OUT.Watchdog_Interrupted or
        SENSOR_OUT.Temp or SENSOR_OUT.Humidity or SENSOR_OUT.Air_Pressure or
        SENSOR_OUT.Air_In_Line or SENSOR_OUT.Occlusion or SENSOR_OUT.Door_Open);

-- when there is no infusion cancel, infusion stop situation does not exist.
property no_cancel_implies_no_stop_conditions =
        not(OP_CMD_IN.Infusion_Cancel) => not(sensor_conditions_that_cause_pause);
assume "no_cancel_implies_no_stop_conditions" :no_cancel_implies_no_stop_conditions;
....
-- If there is no infusion cancel then there is no condition stopping infusion.
property no_cancel_implies_no_stop_conditions =
  not(OP_CMD_IN.Infusion_Cancel) =>
      not(infusion_end_conditions or any_alarms or OP_CMD_IN.Infusion_Inhibit);
assume "no_cancel_implies_no_stop_conditions" :no_cancel_implies_no_stop_conditions ;

-- System will continue to be ACTIVE if no conditions to stop infusion occurs.
property in_active = true -> (pre(GPCA_SW_OUT.Current_System_Mode) >= 2 and
    pre(GPCA_SW_OUT.Current_System_Mode) <= 4) and
        not(OP_CMD_IN.Infusion_Cancel) => (GPCA_SW_OUT.Current_System_Mode >= 2 and
            GPCA_SW_OUT.Current_System_Mode <= 4);
guarantee "in_active":  in_active;
```

Figure 6.9: Portions of AGREE properties at different layers of system abstraction

Finally, in the concrete model, there are more modes of active drug infusion than the ones specified in the abstract model as shown in Figure 6.7. Originally the abstract model's property was, *when the system is started and not stopped, the system infuses at BASAL flow rate (when there is no patient bolus request).* But the concrete model switches between `BASAL` and `OTHER BOLUS` based on the prescription. There were multiple ways to reconcile the differences. A straightforward approach is to assume that there is no prescribed `OTHER BOLUS`. However, this is not a reasonable assumption for the scenarios under consideration – the essential system properties must hold for any infusion scenario. Therefore, we adopted an alternative route: the abstract property was modified to state that "*system infuses at no less than the* `BASAL` *flow rate*". With an assumption that any bolus infusion delivers drug at a rate higher than the `BASAL` flow rate, which is typically true of the problem domain, this change allows the possibility of multiple drug delivery modes with different flow rates even when the abstraction includes only a few specific modes explicitly.

Apart from these changes, some auxiliary properties were required to guarantee that the concrete model's states and transitions that were not represented by the abstract model are not reachable given the system assumptions and the abstraction mapping. For example, the concrete model has an additional state `PAUSED` shown in Figure 6.7. Thus, an auxiliary property to show that `PAUSED` is never reached under the given environmental assumptions is needed.

In summary, the goal of this process is to justify the following claims:

1. Every abstract state, input and output in the abstract infusion pump model has some matching counterparts in the concrete model.

2. States, inputs and outputs of *interest* in the concrete GPCA have some matching counterparts in the abstract model.

3. Transitions between such matched states on matched inputs in the concrete GPCA model have a corresponding matching transition in the abstract model and the respective outputs match.

4. Unmatched concrete inputs do not occur given certain environmental assumptions.

5. Unmatched concrete states are not reachable under those environmental assumptions.

Taken together these claims, informally, allow us to see the abstract infusion pump model as a stand-in (simulation) for the concrete GPCA model under the given environmental assumptions. Thus safety properties that are established for a system model that uses the abstract infusion pump as a component, in a way that satisfies the environmental assumptions made in the matching process, will be upheld when the concrete GPCA infusion pump is substituted in place of the abstract pump.

6.6 Discussion

As a proof of concept of our hierarchical verification approach, we proved a total of 7 `PCA_Pump` properties in AGREE that correspond to the abstract timed automata model of the PCA. In order to guarantee these 7 system properties, there were 7 `GPCA_IP_Interface` properties and 11 `GPCA_SW` properties along with some auxiliary properties of the `GPCA_HW_Sensors`, `GPCA_HW_Actuators` and `GPCA_OP_Interface` to route input and output signals. Similarly the 11 `GPCA_SW` properties were in turn guaranteed by 4 `ALARM` , 11 `IM` and 6 `CONFIG` component properties.

6.6.1 Tool Limitations

Our current approach to verification is restricted in several ways. First, AGREE currently only handles synchronous architectural models in which execution proceeds in a

deterministic discrete sequence of steps. Second, AGREE can verify only *invariants*, so liveness properties, cannot be specified in AGREE. In our experience, this is not as severe a limitation as it may seem, since most systems are concerned with *bounded liveness* in which an action must occur within a time interval that can be written in AGREE.

The current analysis tools use *rationals* to model the behavior of real numbers; However, most software is implemented using floating point numbers. This can lead to unsoundness in our analysis of software that uses floating point arithmetic. Also, AGREE does not support trigonometric or non-linear functions. These can be approximated in some cases, but many of the interesting numeric properties of systems simply cannot be specified.

6.6.2 Summary

When systems are composed from sub-systems, properties of the system must be assessed based on its component properties and the composition. To reason about such systems, no single analysis and modeling method can successfully cope with all aspects of a system or its components. Hence multiple notations and formalisms are used. An approach to logically glue the diverse analysis of the system and its components is required to reason about the system properties.

In this chapter, we considered compositional verification of a medical system at multiple levels of abstraction, with different formalisms used at each level. We were able to semi-formally glue distinct verification paradigms by leveraging the system's hierarchical architectural decomposition. We showed how properties proved for the system components at the lower levels of abstraction can be used to validate the more abstract models, ensuring that properties proved at the higher levels of abstraction remain satisfied.

While techniques used in this chapter are specific to the GPCA, we believe that this work can form the basis for a general, scalable and practical approach to layered verification of properties in complex cyber-physical systems. In order to fully realize the promise of this approach, we have defined a formal hierarchical proof-based approach. It extends the AGREE tools to automate the verification of user-defined simulation relation between two AADL components. This would then enable one to even mechanically translate the abstract model from a different formalism into AADL and then define and verify the simulation relation formally using AGREE. The details of the formal mapping are discussed elsewhere [91].

Chapter 7

Requirements traceability

Conceptual integrity is the most important consideration in systems. – Fred Brooks [9]

In the previous chapters we described how the reference model guides a typical model based requirements engineering approach using the infusion pump case example. In this chapter, we present a way that leverages the foundational concepts defined in the reference model to support advanced requirements analysis, namely *Requirements traceability* that is defined as

> *"the ability to describe and follow the life of a requirement, in both forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)."* [29].

This topic has been of great interest in research and practical relevance in the industry for several decades. Intuitively, it concerns establishing relationships, called *trace links*, between the requirements and one or more artifacts of the system. Among the several different development artifacts and the relationships that be can established

from/to the requirements, being able to establish trace links from requirements to artifacts that realize or *satisfy* those requirements—particularly to entities within those artifacts called *target artifacts* [28]—has been enormously useful in practice. For instance, it helps analyze the impact of changes in one artifact on the other, assess the quality of the system, aid in creating assurance arguments for the system, etc. In fact, certification standards for safety-critical systems (e.g., [18, 79]) usually require traceability matrices that map high-level requirements to lower-level requirements and (eventually) leaf-level requirements. In this section, we focus our attention to this subset of requirement traceability, that we call *Requirements Satisfaction Traceability.*

Recalling the discussion of hierarchical requirements satisfaction from our reference model in Chapter 3, given an architectural decomposition of a component, it establishes a relationship between the requirements of the sub-components in the decomposition and the component's requirements that they satisfy. Instead of just recording the trace links using manual mappings [44] or heuristic approach involving natural language processing [48], Hierarchical Requirements Satisfaction offers a semantically rich way to establish them. Originally proposed by Zave and Jackson [93], the satisfaction relationship between requirements demonstrates how a set of requirements is being met by the composed behaviors of components, such as the relation between the user's requirements and the composition of the system and its environment that satisfy them.

From a traceability perspective, these satisfaction arguments help establish trace links (the *satisfied by* relationship) between the requirements and those parts of the system and environment (the target artifacts) that were necessary to satisfy the requirements; We call those target artifacts a *set of support* for that requirement. Mathematically, if we think of the argument as a proof in which the requirement is the claim, then the set of support is the set of axioms (or clauses) that were necessary to prove the claim, and the trace links are means to associate the claim to those clauses. Such proofs

are not, in general, unique, and often there are multiple sets of clauses that could be used to construct a proof. The existing traceability literature does not discuss multiple alternative satisfaction arguments or sets of trace links for one system design.

In the context of satisfaction arguments and *satisfied by* trace links, it is beneficial to distinguish between trace links to *"a"* set of support (containing the clauses needed to make a satisfaction argument) vs. *"the"* sets of support (all clauses needed to make all possible satisfaction arguments). Establishing trace links to all sets of support, that we call *complete* traceability, provides insight into the elements of the set of support—elements that are necessary for any satisfaction argument, elements that are needed for some satisfaction arguments, and elements that do not contribute to the satisfaction of the requirement of interest. We categorize the elements as *Must*, *May* and *Irrelevant* support elements for each requirement.

In the rest of this chapter, we examine how the notion of using hierarchical satisfaction relationship to establish completeness in traceability changes the way traceability is perceived, established, maintained, and used. We also introduce and discuss the notion of completeness in traceability, which considers **all** *satisfied by* trace links between a requirement and the target artifacts that work to satisfy the requirement, and contrast this notion with the partial traceability common in practice.

7.1 An Example

Consider the GPCA system that infuses drug only if the patient's vital signs are normal and not otherwise. While developing and maintaining such safety critical systems, establishing traceability between the requirements to parts of the system that work to satisfy the requirements is crucial to analyze and assess the impact of change of one on the other. For example, assume one of the device's requirements is to *raise an alarm if the patient's vitals is lower than a certain threshold, T1.* For this requirement, a trace

link is established to a sensor that (*measures the patient's vitals and reports a hazard if it is lower than T1*), a LED that (*produces a blinking red light if the sensor reports the hazard*), and an assumption (*blinking red light from LED and will attract attention*)—three artifacts that when working together satisfy the requirement. These trace links help identify the components that are impacted by a change in the requirement and vice versa. For instance, if the device is now expected to raise an alarm if the vitals lower than another threshold (say, less than T1), the trace links established helps identify that the sensor functionality needs to be changed to meet the new requirement.

However, if there was an alternate way the original requirement could be satisfied that was not explicitly traced, then the inferences from the impact analysis may be misleading. For instance, if the device had two redundant sensors for fault tolerance, then the device might continue to raise alarms when a patient's vitals drops below T1 based on the report from the unchanged sensor system. Further, components can be involved in the satisfaction of more than one requirement. If we are unable to trace back to all the requirements those components help satisfy, we may not be able to adequately assess the impact of changes. For example, if the hazard output from the modified sensor was used by another connected medical device, then the impact of changing that sensor should also be traced back to the requirements on the other system.

In fact, actual systems often have hundreds of system requirements and numerous components and, for many requirements, several possible satisfaction arguments. To be able to perform precise impact analysis, it is crucial to identify trace links between the requirements and all the possible ways the system can satisfy the requirements. Unfortunately, to the best of our knowledge, neither the completeness of traceability nor the impact of incomplete traceability are discussed in the existing literature.

7.2 Formal Representation of Traceability

In its simplest sense, traceability is capturing the relationship between artifacts. There are three building blocks for traceability—a *source artifact* from which relationships should be established, *the target artifact* to which the source artifact is to be associated, and the *trace relationship* that describes the association between the artifacts. To provide notation, we write $T \mid\!\rightsquigarrow e$ to state that a set of target artifacts T is sufficient to establish a traceability relationship $\mid\!\rightsquigarrow$ to source element e.

For requirements satisfaction traceability, the target artifacts are set of requirements Δ, the set of source artifacts Σ contains the elements that realize the requirement such as lines of code, design elements, system or world assumptions, or lower-level requirements, and the trace relationship is *satisfies*. Thus, we write $S \mid\!\rightsquigarrow_s r$ for $S \subseteq \Sigma$ and $r \in \Delta$ when set of target elements S contributed to satisfying[1] requirement r.

We assume that the satisfaction deduction $\mid\!\rightsquigarrow_s$ is monotonic on the subset relation over Σ, that is, if $S' \subseteq \Sigma$, $S \subset S'$ and $S \mid\!\rightsquigarrow_s r$, then $S' \mid\!\rightsquigarrow_s r$.

The monotonicity of the satisfaction relation means that, unless *all* elements of the implementation Σ are required for a proof, there are multiple implementation sets $S \subset S' \subset \ldots \subset \Sigma$ that can satisfy a given requirement r. However, we are primarily interested in *minimal* sets that satisfy r; tracing a requirement to the entire implementation is not particularly enlightening. We call a minimal set of target artifacts a *set of support* for that requirement, and define the SOS relation to associate sets of support to requirements.

$$SOS(r, S) \equiv S \mid\!\rightsquigarrow_s r \ \wedge (\neg \exists S' \ . \ S' \subset S \wedge S' \mid\!\rightsquigarrow_s r) \tag{7.1}$$

[1]The $\mid\!\rightsquigarrow_s$ introduced here establishes the satisfaction based traceability relationship between individual constraints and predicates, whereas the $\vdash$ relationship defined in Section 3 is a relationship established between runs (the interpretation of predicates and constraints). While traceability relationship can be expressed using $\vdash$, we choose to define this new relationship to keep the explanation simple.

SOS maps sets of support to a requirement. As mentioned earlier, there could be many sets of support for a requirement. To capture that notion, we define, *all sets of support (ASOS)* for a requirement as an association to all its sets of support.

$$ASOS(r) \equiv \{\ S | S \subseteq \Sigma \wedge (r, S) \in SOS\ \} \tag{7.2}$$

The set of *ASOS*-es for all requirements represents the complete traceability of the system.

7.3 Complete Traceability

Through formal definitions in the previous section we highlight two critical facets – the semantics and completeness – in establishing traceability. In practice, just being able to trace a requirement to a target artifact that satisfies it, say a line of code, is less useful [32]; a holistic view of how that line of code in conjunction with other related lines of code satisfy the requirement provides meaningful information to perform analysis such as assessing the impact of a change [17]. By defining a trace from requirements to the set of target artifacts, we advocate that traceability be captured in a way that upholds its semantic rationale. Further, we also found that performing analysis, such as impact analysis, using a partial set of trace links may result in imprecise results and misplaced confidence about the system. By considering *SOS* trace links and complete traces for each requirement, we can assess analyses related to requirements satisfaction traceability on a proper semantic foundation. In this section, we elaborate on how the semantic foundations in the previous section help us understand, assess, and use traceability precisely.

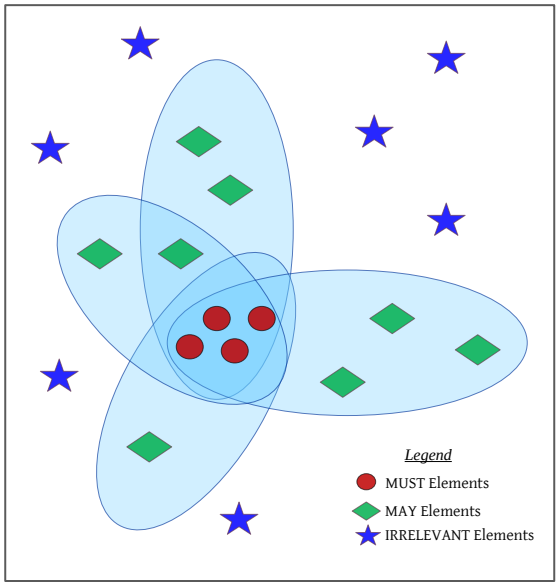


Figure 7.1: May - Must - Irrelevant Set of Support

7.3.1 Categorizing the Set of Support

Establishing *ASOS* for a requirement, one gets a clear picture of the all possible ways that requirement is satisfied. This information helps categorize each target artifact into one of the following groups for that requirement. The relationships are illustrated graphically in Figure 7.1, and explained formally below.

- **MUST** elements - target artifacts that are present in all the sets of support for a requirement (red circles in Figure 7.1).

$$MUST(r) = \bigcap ASOS(r) \tag{7.3}$$

- **MAY** elements - target artifacts that are used in some, but not all, sets of support

(green diamonds in Figure 7.1).

$$MAY(r) = (\bigcup ASOS(r)) \setminus MUST(r) \tag{7.4}$$

- **IRRELEVANT** elements - target artifacts that are not in any of the set of support (blue stars in Figure 7.1).

$$IRR(r) = \Sigma \setminus (\bigcup ASOS(r)) \tag{7.5}$$

This categorization helps identify the role and relevance of each target artifact in satisfying a requirement. The MUST elements are those target artifacts that are absolutely necessary for the requirement satisfaction. Hence, any change to these elements will most likely impact on each other. On the other hand, MAY elements indicate those target artifacts that satisfy the requirement in one of the possible ways. Any change to just one of these elements will not affect the satisfaction of that requirement. The IRRELEVANT elements are never required to satisfy the requirement, so neither does a change in these artifacts affect the satisfaction of the requirement, nor does a change in the requirement necessitate a change in these elements (at least in terms of satisfaction).

7.3.2 Using Traces for Precise Analysis

The categorization of the set of support to be useful in several analyses:

Impact Analysis The *ASOS* set improves understanding of how a change in the requirement will affect the target artifacts and vice versa. While the *ASOS* of a requirement gives a clear picture of various ways a requirement is satisfied by the system, the categorization of target artifacts helps precisely assess and plan when and where the changes have to be implemented. The *MUST* elements are those target artifacts that

are highly likely to change with any change in the requirement, whereas not all MAY elements may need to be changed.

If a requirement has elements only in its MAY set, that is if $MUST$ set is empty ($MUST(r) = \emptyset$), it indicates that the requirement has been (intentionally or unintentionally) implemented in independent ways, such as fault tolerant systems. For such requirements, one has to carefully analyze and decide if the target artifacts in all or one disjoint set needs to be changed. These analysis could be performed either from the perspective of one or all requirements of the system.

From the target artifact side, this categorization helps analyze the impact of changes to the artifact. Suppose we decide to change a target artifact in the MAY set for a requirement. While one might think that it is safe to change this artifact since it does not affect that requirement's satisfaction, an examination of the $ASOS$ sets of other requirements helps identify if it is indeed safe to change that artifact. If it is present in the $MUST$ set for another requirement, then a change to this artifact will definitely impact the other requirement. However, if it is in the MAY sets for all the requirements, then it is clearly safe to change. Hence, this categorization helps us to assess critical dependencies between the target artifacts and the satisfaction of requirements and thus enables a precise bi-directional impact analysis of a change.

Verification and Validation Complete traceability can assist in tailoring verification and validation in systems. For instance, if several requirements have a certain target artifact in their $MUST$ set, say an particular assumption, it reveals the importance of focusing V&V attention on that artifact. Along the same lines, for a system with a complex architecture (components that each have functionality) such as system of systems, this categorization helps identify components that is critical to satisfy most requirements. This categorization helps plan verification strategies.

If we examine changing a target artifact that appears $MUST$ set for any requirement, then this requirement must be re-verified. However, if it appears in the MAY set for the requirement, then we can instead remove any sets-of-support that contain the element; as long as there still exists at least one set of support for the requirement, no re-verification is necessary for that requirement.

Further, the notion of complete traces helps to assess if requirements are satisfied by the system in an unintended manner. It is well known that issues such as vacuity [49] can cause requirements to be satisfied in a trivial manner. Even for non-vacuous requirements, it can be the case that requirements can be satisfied using a much smaller portion of the system than intended because they are incorrectly specified. By capturing all the sets of support and categorizing the items therein, it is possible to examine whether the MUST set corresponds to expectations on the system, or, if more rigor is required, to examine each set of support individually to see whether it matches the expectations of the architects and developers.

Completeness Checking By getting all the sets of support for all requirements of the system and categorizing them, one can get a clear picture about the role of target artifacts in satisfying the requirement(s). By reversing the direction of the complete requirement satisfaction trace, one can find if there are target artifacts that do not trace to any requirement. This can be performed by examining the minimal set of target elements used by *any* SOS for *all* requirements, in other words the set $\bigcap\{IRR(r)|r \in \Delta\}$. If this set is non-empty, it is a possible indication of "gold plating" or missing requirements. In other words, it helps assess if the requirements of the system describe all the behaviors of the system. Being able to assess the coverage of requirements over the model is crucial in the safety critical system domain.

Benchmarking in Traceability There is substantial interest within the Requirements Engineering research community (and our immediate research group) towards automating the construction and maintenance of traceability links [38, 19, 11]. To that end, there are repositories such as the Data sets published at Center of Excellence for Software Traceability [12] containing many example systems, each with a reasonably complete set of requirements and target artifacts and with trace links constructed by groups of experts. It is then possible to benchmark automated and semi-automated traceability approaches against vetted sets of trace links.

We are, in general, highly supportive of this approach: benchmarking against a reasonable set of candidate models has been very important in several areas of computer science research, including CPU [43], GPU [87], and compiler performance [3], performance of SAT/SMT solvers and model checkers [86], and many other areas. However, it can also be misleading if the benchmark problems are not carefully selected or if results are incorrectly computed, as described in, e.g., several articles critical of benchmarking for CPU/GPU performance [21, 78, 77].

For traceability research, the usual standard measures for examining the performance of different approaches is in terms of *precision* and *recall* against the "gold standard" set of traceability links that exist in repositories. Our concern is that, for requirements satisfaction traceability, there are often many such sets of valid links, as we have explored in this paper, so these metrics may be misleading. One can envision situations in which the gold standard pursues one set of support and the automated approach pursues another, leading to low precision and recall scores. A close examination of traceability links and categorizations such as the ones we have explored may be useful to provide more accurate measurements of the quality of automated approaches. If requirements have more than one SOS, the gold standard should probably contain each of these arguments. In addition, the ability to characterize trace links as MAY or $MUST$ may

be a fruitful direction of future research for automated approaches to constructing trace links.

7.4 Prototype Implementation

We have implemented some of these ideas in a branch of the AGREE/AADL tool suite [2, 67, 13], that determine whether safety properties hold of complex finite or infinite-state systems modeled in AADL and AGREE. While we called the trace elements as the set of support in our original formulation of traceability, in the AGREE implementation it was renamed as *inductive validity core* (IVC).

Our branch of the AGREE tools provides traceability information, the IVC explains proofs produced by inductive model checkers, in much the same way that a counterexample explains negative results. In other words, IVCs offer an explanation as to why a property is satisfied by a model in a formal and human-understandable way. Informally, if a decomposition is viewed as a conjunction of sub-component requirements and architectural connections, an IVC is a set of those sub-component requirements that are sufficient to construct a proof such that if any constraint is removed, the property is no longer valid.

IVC implementation in AGREE is influenced by earlier work on UNSAT cores [94] that provide the same kind of information for individual SAT or SMT queries, and this approach has been lifted to bounded analysis for Alloy in [83]. What we provide is a generic and efficient mechanism for extracting supporting information, similar to an UNSAT core, from the proofs of safety properties using inductive techniques such as PDR and k-induction. Because many properties are not themselves inductive, these proof techniques introduce lemmas as part of the solving process in order to strengthen the properties and make them inductive. Our technique allows efficient, accurate, and precise extraction of inductive validity cores even in the presence of such auxiliary

lemmas.

While the detailed definition and implementation of IVC algorithms is out of scope of this dissertation and is discussed independently elsewhere [27], in the rest of this section, we illustrate how we leverage it to establish traceability using the GPCA

7.4.1 Traceability of GPCA

As mentioned in earlier chapters that the verification of GPCA software's architectural decomposition was performed using AGREE tools, we leverage the IVCs implementation in AGREE to establish traceability. As shown in Figure 7.2, we derived the set of support for GPCA software requirements.

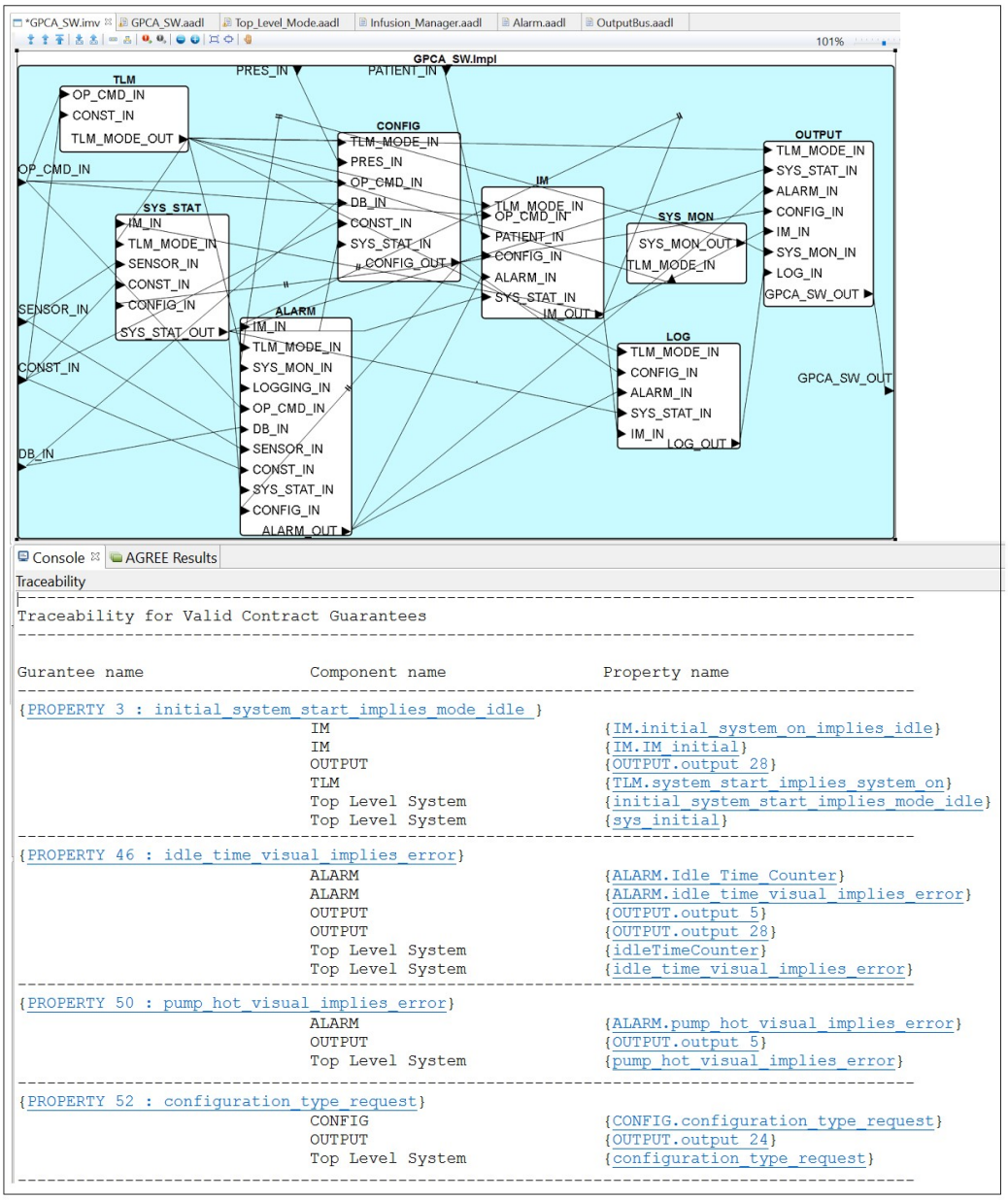


Figure 7.2: GPCA Traceability

7.5 Discussion

While there has been a large amount of work in the area of requirements traceability, there are no definitions of what constitutes a "good" set of trace links, nor are there any well defined metrics that can be used to measure the quality of a set of trace links. This, of course, raises questions as to how to reason about the fit for purpose of a set of trace links, how to compare sets of trace links, how to judge the quality of automate trace link generation (what do we compare to and how do we do it?), and other concerns.

In this chapter we define a theoretical framework that can be used to understand and assess requirements satisfaction traceability—the trace links from a requirement to its implementation. By defining a requirements satisfaction trace from a requirement to a set of support we have a clear definition of *sufficient* set of trace links—a set that identifies all implementation and environment elements needed to make *one* satisfaction argument. Similarly, we can discuss a *complete* set of trace links—the set that identifies all implementation and environment elements needed to make *all* satisfaction arguments. It is worth reiterating that our goal is not to provide another formalization of traceability, but to provide the foundation for future research related to satisfaction trace link generation, analysis, and assessment.

Our thinking related to sets of support, and sufficient and complete sets of trace links stem from our desire to automatically generate such trace links in the context of model-based software development. While one may think that it is impossible—or at least extraordinarily difficult—to identify all trace link in practice [80], some of the initial results based on our work in formal verification indicate that such complete requirements satisfaction traces can be established efficiently for substantial models [27].

In the future, we hope to provide a computational framework to allow an analyst to reason about the trace links to, for example, identify the components in an architecture containing many *must* elements and flag those components for extensive verification,

and to identify disjoint sets of support to explore the fault tolerant properties of an architectural solution.

Chapter 8

Conclusion and Future Work

Specifying and analysing the requirements of safety critical systems is complicated due to the hierarchy of subsystems used to build it; a hierarchy that necessitates requirements flow-down to the architectural components the system is decomposed into as well as requirements "flow-up" as the architectural decomposition efforts help illuminate the desired system behavior. Consequently requirements related issues left unnoticed during development have lead to major accidents. With the increasing need to develop more and more complex systems, there is a strong need to improve requirements engineering of such systems.

The first step towards such an improvement is to clarify the conceptual confusions and uncertainties that exists in the basic concepts in requirements engineering of complex systems. Since such concepts, in part, have already been captured in the form of reference models, we have refined and extended the existing models to to address the needs we have identified. In general, reference models serve as a frame of reference that one can easily relate to any real system in a conceptually clear manner and reason about the system's requirements. However, since the existing models (defined in the 90s) did not adequately account for the complexity in system design and its influence

on the requirement, they are now inadequate to be used as reference in modern system developments are typically constructed as a hierarchical federation of components.

The proposed Hierarchical reference model provides a generic and simple framework to represent and reason requirements of such systems. By generalizing the notion of architecture and capturing its relationship with the requirements, this model addresses the inadequacies of the existing models to be used for system with complex hierarchical architectures. The properties specified in the model helps system and software engineers to rigorously reason about the correctness of the requirements. We also demonstrated the usefulness of the reference model using a GPCA system. While the choice of notion, tools, and techniques are specific to the GPCA, we believe that the approaches and the artifacts shall serve as an exemplar for the requirements engineering community.

Further, we defined a new formal definition of traceability based on the hierarchical satisfaction relationship. This way of formally defining traceability allowed us to formulate the notion of "complete traceability"—the ability to identify all possible satisfaction trace links between requirements. Further, it helped bring out the distinction between "necessary" vs. "sufficient" sub-systems' requirements to achieve a system-level requirement. This new insight into traceability, we believe, will greatly help engineers manage requirements of large complex systems.

Nevertheless, there is still work to be done and we aim to pursue the following areas for exploration in the near future.

- Defining methodologies that address capturing requirements and refining them hierarchically. While the reference model provides a theoretical basis to understand and reason about requirements, putting them to practice is a significant engineering task. While the GPCA illustration provides an outline of a method to perform requirements analysis, one of the areas to explore is defining and validating an end-to-end methodology that will systematically guide engineers to define,

refine, and reason about requirements and architectures.

- Exploring requirements in the stochastic domain. One of the challenging future directions is exploring requirements at higher levels of system abstraction, an abstraction level where the system interacts with the environment and physical components whose behaviours are stochastic in nature. However, specifying stochastic requirements is not adequately explored in the requirements engineering community. To that end, we have attempted to use probabilistic modeling and verification techniques and a tool called PRISM [50] to identify requirements. Since the results of our initial exploration is promising [68], we wish to explore the area of stochastic requirements, especially in the safety critical domain.

- Acceptability-based Traceability. Testing remains a widely used technique to check conformance of the component behaviours to their requirements. Consequently, the need to establish traceability relationship between the requirements and its components in the testing context is imperative. Yet, it has not be adequately explored in this domain. However, the formal definition of "acceptability" defined in the reference model serves as a basis to explore and define rigorous traceability relationship.

Bibliography

[1] Anaheed Ayoub, BaekGyu Kim, Insup Lee, and Oleg Sokolsky. A systematic approach to justifying sufficient confidence in software safety arguments. In *International Conference on Computer Safety, Reliability, and Security*, pages 305–316. Springer, 2012.

[2] John Backes, Darren Cofer, Steven Miller, and Michael W Whalen. Requirements analysis of a quad-redundant flight control system. In *NASA Formal Methods Symposium*, pages 82–96. Springer, 2015.

[3] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks summary and preliminary results. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165. IEEE, 1991.

[4] Barry Boehm. Requirements that handle ikiwisi, cots, and rapid change. *Computer*, 33(7):99–102, 2000.

[5] Boeing. 747 fun facts. http://www.boeing.com/commercial/ 747family/pf/pf_facts.html.

[6] Jody Lannen Brady. First, do no harm: making infusion pumps safer. *Biomedical instrumentation & technology*, 44(5):372–380, 2010.

[7] Manuel Brandozzi and Dewayne E Perry. Transforming goal oriented requirement specifications into architecture prescriptions. In *Workshop from Soft. Req. to Arch*, pages 54–61, 2001.

[8] F Brooks and HJ Kugler. *No silver bullet*. April, 1987.

[9] Frederick P Brooks. The mythical man-month, 1975.

[10] Robert Charette. This car runs on code. *IEEE Spectrum*, February 2009.

[11] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35, 2007.

[12] CoEST. Center of excellence for software traceability. `http://www.CoEST.org`. Accessed: 2019-10-30.

[13] Darren Cofer, Andrew Gacek, Steven Miller, Michael W Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In *NASA Formal Methods Symposium*, pages 126–140. Springer, 2012.

[14] Steve Mc Connell. Quotes on requirements. http://wiki.c2.com/?QuotesOnRequirementsAndUsers.

[15] Jeremy Dick. Design traceability. *IEEE software*, 22(6):14–16, 2005.

[16] Jeremy Dick and Jonathon Chard. 7.1.2 the systems engineering sandwich: combining requirements, models and design. *INCOSE International Symposium*, 14(1):1401–1414, 2004.

[17] Jeremy Dick, Elizabeth Hull, and Ken Jackson. *Requirements engineering*. Springer, 2017.

[18] RTCA DO. 178c, software considerations in airborne systems and equipment certification. rtca. *Inc., Washington, DC, USA (December 1992)*, 2011.

[19] Alexander Egyed and Paul Grunbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, pages 163–171. IEEE, 2002.

[20] EU. Speculative and exploratory design in systems engineering. `https://cordis.europa.eu/project/rcn/79466/factsheet/en`. 2019.

[21] ExtremeTech. Driver irregularities may inflate nvidia benchmarks. `http://www.extremetech.com/computing/54154-driver-irregularitiesmay-inflate-nvidia-benchmarks`. 2003.

[22] FDA. Computer world. `https://www.computerworld.com.au/article/538113/toyota_recalling_nearly_2_million_priuses_due_software_glitch/`. Accessed: 2017-09-24.

[23] Food, Drug Administration, et al. White paper: infusion pump improvement initiative, 2010.

[24] US Food, Drug Administration, et al. Guidance for industry and fda staff-total product life cycle: Infusion pump-premarket notification [510 (k)] submissions. *Issued April*, 23, 2010.

[25] US Food and Drug Administration. Infusion Pump Improvement Initiative. https://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSu

[26] US Food, Drug Administration, et al. Infusion pumps total product life cycle: guidance for industry and fda staff. *Food and Drug Administration Standard*, pages 910–766, 2014.

[27] Elaheh Ghassabani, Andrew Gacek, and Michael W Whalen. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 314–325. ACM, 2016.

[28] Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. *Traceability fundamentals*, pages 3–22. Springer, 2012.

[29] Orlena CZ Gotel and CW Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101. IEEE, 1994.

[30] Paul Grunbacher, Alexander Egyed, and Nenad Medvidovic. Reconciling software requirements and architectures: The cbsp approach. In *Proceedings of 5th Int'l Symposium on Requirements Engineering*, pages 202–211. IEEE, 2001.

[31] Carl A Gunter, Elsa L Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[32] Jin Guo, Natawut Monaikul, and Jane Cleland-Huang. Trace links explained: An automated approach for generating rationales. In *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, pages 202–207. IEEE, 2015.

[33] Jon G Hall, Michael Jackson, Robin C Laney, Bashar Nuseibeh, and Lucia Rapanotti. Relating software requirements and architectures using problem frames.

In *Proceedings of Joint Int'l Conf. on Requirements Engineering*, pages 137–144. IEEE, 2002.

[34] Jon G Hall and Lucia Rapanotti. A reference model for requirements engineering. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 181–187. IEEE, 2003.

[35] Jonathan Hammond, Rosamund Rawlings, and Anthony Hall. Will it work?[requirements engineering]. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 102–109. IEEE, 2001.

[36] Terry L Hardy. *Software and System Safety*. AuthorHouse, 2012.

[37] Michael D Harrison, Paolo Masci, José Creissac Campos, and Paul Curzon. Demonstrating that medical devices satisfy user related safety requirements. In *Software Engineering in Health Care*, pages 113–128. Springer, 2014.

[38] Jane Huffman Hayes, Alex Dekhtyar, and James Osborne. Improving requirements tracing via information retrieval. In *Proceedings. 11th IEEE International Requirements Engineering Conference, 2003.*, pages 138–147. IEEE, 2003.

[39] Mats Heimdahl, Lian Duan, Anitha Murugesan, and Sanjai Rayadurgam. Modeling and requirements on the physical side of cyber-physical systems. In *Second Int'l Workshop on the Twin Peaks of Requirements and Architecture*, May 2013.

[40] Constance Heitmeyer, Ralph Jeffords, Ramesh Bharadwaj, and Myla Archer. Re theory meets software practice: Lessons from the software development trenches. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 265–268. IEEE, 2007.

[41] Constance L Heitmeyer and Ramesh Bharadwaj. Applying the scr requirements

method to the light control case study. *Journal of Universal Computer Science*, 6(7):650–678, 2000.

[42] Constance L Heitmeyer, Ralph D Jeffords, and Bruce G Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231–261, 1996.

[43] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[44] MathWorks Inc. Simulink requirements traceability. `https://www.mathworks.com/discovery/requirements-traceability.html`. Accessed: 2019-10-30.

[45] Michael Jackson. The world and the machine. In *Proceedings of the 17th international conference on Software engineering*, pages 283–292. ACM, 1995.

[46] R Jetley and P Jones. Safety requirements based analysis of infusion pump software. *IEEE RTSS/SMDS*, pages 310–325, 2007.

[47] Anjali Joshi, Steven P Miller, and Mats PE Heimdahl. Mode confusion analysis of a flight guidance system using formal methods. In *Proceedings of 22nd Digital Avionics Systems Conf.(DASC'03)*, volume 1, pages 2–D. IEEE, 2003.

[48] Ed Keenan, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, et al. Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1375–1378. IEEE, 2012.

[49] Orna Kupferman and Moshe Y Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer*, 4(2):224–233, 2003.

[50] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[51] Brian R Larson, John Hatcliff, and Patrice Chalin. Open source patient-controlled analgesic pump requirements documentation. In *Proceedings of the 5th International Workshop on Software Engineering in Health Care*, pages 28–34. IEEE Press, 2013.

[52] Nancy Leveson. *Engineering a safer world: Systems thinking applied to safety*. MIT press, 2011.

[53] Nancy Leveson, L. Denise Pinnel, Sean David Sandys, Shuichi Koga, and Jon Damon Reese. Analyzing software specifications for mode confusion potential. In *Proceedings of a Workshop on Human Error and System Development*, pages 132–146, 1997.

[54] Nancy G Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on software engineering*, 26(1):15–35, 2000.

[55] Nancy G Leveson and Jorge Diaz-Herrera. *Safeware: system safety and computers*, volume 680. Addison-Wesley Reading, 1995.

[56] Robyn R Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 126–133. IEEE, 1993.

[57] Luiz Eduardo G Martins and Tony Gorschek. Requirements engineering for safety-critical systems: A systematic literature review. *Information and software technology*, 75:71–89, 2016.

[58] Paolo Masci, Yi Zhang, Paul Jones, Harold Thimbleby, and Paul Curzon. A generic user interface architecture for analyzing use hazards in infusion pump software. In *5th Workshop on Medical Cyber-Physical Systems*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[59] MathWorks Inc. Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier.

[60] MathWorks Inc. Products. http://www.mathworks.com/products.

[61] S. P. Miller, M. P.E. Heimdahl, and A.C. Tribble. Proving the shalls. In *Proceedings of FM 2003: the 12th Int'l FME Symposium*, September 2003.

[62] Steven P Miller and Alan C Tribble. Extending the four-variable model to bridge the system-software gap. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 4E5–1. IEEE, 2001.

[63] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *Int. J. Softw. Tools Technol. Transf.*, 8(4):303–319, 2006.

[64] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

[65] Anitha Murugesan, Sanjai Rayadurgam, and Mats Heimdahl. Modes, features, and state-based modeling for clarity and flexibility. In *Fifth Int'l Workshop on Modeling in Software Engineering*, May 2013.

[66] Anitha Murugesan, Sanjai Rayadurgam, and Mats Heimdahl. Using models to address challenges in specifying requirements for medical cyber-physical systems. In *Fourth workshop on Medical Cyber-Physical Systems*. Citeseer, 2013.

[67] 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.

[68] Anitha Murugesan, Michael W Whalen, Neha Rungta, Oksana Tkachuk, Suzette Person, Mats PE Heimdahl, and Dongjiang You. Are we there yet? determining the adequacy of formalized requirements and test suites. In *NASA Formal Methods Symposium*, pages 279–294. Springer, 2015.

[69] Teryl K Nuckols, Anthony G Bower, Susan M Paddock, Lee H Hilborne, Peggy Wallace, Jeffrey M Rothschild, Anne Griffin, Rollin J Fairbanks, Beverly Carlson, Robert J Panzer, et al. Programmable infusion pumps in icus: an analysis of corresponding adverse drug events. *Journal of General Internal Medicine*, 23(1):41–45, 2008.

[70] Bashar Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–119, 2001.

[71] World Health Organization. Information sheet on opioid overdose. `https://www.who.int/substance_abuse/information-sheet/en/`. 2018.

[72] osate.org. Osate (open source aadl tool environment). `https://osate.org/`. 2019.

[73] M. Pajic, R. Mangharam, O. Sokolsky, D. Arney, J. Goldman, and I. Lee. Model-driven safety analysis of closed-loop medical systems. *Industrial Informatics, IEEE Transactions on*, PP:1–12, 2012. In early online access.

[74] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer programming*, 25(1):41–61, 1995.

[75] Jens Rasmussen and Morten Lind. Coping with complexity. *HG Stassen (Ed.)*, 1981.

[76] SAE. Architecture analysis and design language. `http://www.aadl.info/aadl/downloads/papers/AADLLanguageSummary.pdf`. 2019.

[77] Mohamed Sayeed, Hansang Bae, Yili Zheng, Brian Armstrong, Rudolph Eigenmann, and Gaisal Saied. Measuring high-performance computing with real applications. *Computing in Science & Engineering*, 10(4):60, 2008.

[78] A Lal Shimpi and Brian Klug. They're (almost) all dirty: The state of cheating in android benchmarks, 2013.

[79] Defence Standard. Requirements for safety related software in defence equipment part 2: Guidance.

[80] Darijus Strašunskas. Traceability in collaborative systems development from lifecycle perspective. In *Proceedings of the 1st International Workshop on Traceability*, pages 54–60, 2002.

[81] PG Tate. Model based requirements elicitation. In *3rd IET Int'l Conf. on System Safety*, pages 1–5. IET, 2008.

[82] Jeffrey M Thompson, Mats PE Heimdahl, and Debra M Erickson. Structuring formal control systems specifications for reuse: Surviving hardware changes. In *NASA CONFERENCE PUBLICATION*, pages 117–128. NASA; 1998, 2000.

[83] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *International Symposium on Formal Methods*, pages 326–341. Springer, 2008.

[84] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001.

[85] Axel Van Lamsweerde. From worlds to machines. *A Tribute to Michael Jackson*, 2009.

[86] Miroslav N Velev. Superscalar suite 1.0. *URL: http://www. ece. cmu. edu/ mvelev*, 1999.

[87] V Volkov and JW Demmel. Benchmarking gpus to tune dense linear algebra. 2008. In *Proceedings of the International Conference for High Performance Computing (SC'08), Networking, Storage and Analysis. ACM*, pages 1–11, 2008.

[88] Jonas Westman and Mattias Nyberg. Specifying and structuring requirements on cyber-physical systems using contracts, 2014.

[89] Michael W Whalen, Andrew Gacek, Darren Cofer, Anitha Murugesan, Mats PE Heimdahl, and Sanjai Rayadurgam. Your" what" is my" how": iteration and hierarchy in system design. *IEEE software*, 30(2):54–60, 2013.

[90] Michael W. Whalen, Andrew Gacek, Darren Cofer, Anitha Murugesan, Mats P.E. Heimdahl, and Sanjai Rayadurgam. Your what is my how: Iteration and hierarchy in system design. *Software, IEEE*, 30(2):54–60, 2013.

[91] 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 *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 90–95. IEEE, 2015.

[92] E.S.K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the Third IEEE Int'l Symposium on Requirements Engineering*, pages 226–235. IEEE, 1997.

[93] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30, 1997.

[94] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. *SAT*, 3, 2003.

[95] Didar Zowghi and Vincenzo Gervasi. The three cs of requirements: consistency, completeness, and correctness. In *International Workshop on Requirements Engineering: Foundations for Software Quality, Essen, Germany: Essener Informatik Beitiage*, pages 155–164, 2002.