

Survey of Concerns in Embedded Systems Requirements Engineering

Daniel Aceituna
DISTek Integration, Inc.

ABSTRACT

Requirements engineering is an important part of the software development process. There are various challenges associated with the elicitation, analysis, documentation, and validation of requirements, and these challenges can be more pronounced in embedded software systems, where a number of characteristics, unique to these systems, must be addressed. This paper will examine the unique concerns surrounding the requirements engineering of embedded software systems, including the need to elicit, specify, and validate, interrupts and exceptions, concurrency, and timing. We will then survey and examine some of the current requirements engineering research that can address these concerns. Finally, we will consider whether some of the requirements engineering concerns and practices conceived for, and utilized in the Information Technology (IT) domain, can be beneficial to the development of embedded systems.

CITATION: Aceituna, D., "Survey of Concerns in Embedded Systems Requirements Engineering," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 7(1):2014, doi:10.4271/2013-01-2403.

INTRODUCTION

While all activities in the software development life cycle should be carefully performed to build high quality and reliable software, it is well recognized that requirements engineering is the most important and critical phase to achieving success [1, 5]. Catching an error in the later stage of software development is usually more time consuming and costly than if the error is caught in the early stage [2, 18, 21, 20]. Researchers Boehm and Basili [3] indicate that finding and correcting defects after the product is released is often 100 times more expensive than correcting them during the requirements phase. Table 1 shows the relative cost to address defects in development phases downstream from the requirements [4]. Yet in spite of its importance, the requirements engineering phase of any given project is often the most neglected. One survey performed by Hofmann and Lehner [5] found that successful IT projects spend about 28 times the effort on the requirements phase (compared to unsuccessful projects), indicating that not enough attention is normally dedicated to this important development phase. This lack of attention is not limited to the IT world, but can be found in the embedded software industry as well.

The problem can stem from non-favorable perceptions about requirements [47]. An informal survey of software engineering students, by the author, while teaching a

requirements class, found that none of the students had any intention of focusing their future career in the area of requirements engineering.

Table 1. Relative cost to repair faults in the different software development phases

Development Phase	Cost Ratio
Requirements	1
Design	3-6x
Implementation	10x
Development Testing	15-40x
Acceptance Testing	30-70x
Post Release	40-1000x

Some of the reasons given were "I'm not going to school to learn how to write documents, but how to write code." Another student expressed that they would rather spend the day "Dealing with a computer than dealing with people" [as in stakeholders, and customers]. These perceptions can sometimes extend beyond graduation into industry.

Engineering will sometimes view the requirements document as a living document subject to "many changes to come." That being the case, some may question whether it is productive to spend time compiling, writing, and reviewing something that will need to be revised again and again. Project management will sometimes perceive that progress is not being made until the system enters the design and

implementation phase [47]. Subsequently, projects are often budgeted with more time and money allocated to software implementation and testing than to the requirements phase [48].

Part of what drives the desire to “Start coding as soon as possible” is that people tend to become engineers because they want to solve problems; they prefer to work in the solution domain. This is where the act of designing and coding takes place. Requirements engineering, by contrast, operates in the problem domain [40]; its goal is to define and understand the problem thoroughly enough to produce the best-fit solution.

Due to a human trait known as *Optimistic Bias* [6] there is often a tendency to feel that a problem has been understood well enough to start making progress on the solution. However, less time spent in the problem domain, means less time spent understanding the problem

There is also an interesting observation made by renowned computer scientist Frederick P. Brooks [7, 8]; he states something that all engineers know from firsthand experience, but often fail to consider due to their optimistic bias. The lesson learned from examining many projects is that we can't truly understand the problem we are solving until we actually start working on the solution [8]. The human mind needs to see a solution's attempt at addressing a problem, in order to overcome the optimistic bias and get a real sense of the problem's true difficulty level and the challenges that need to be addressed.

This suggests that we can't fully define requirements until we start designing. This may seem to justify the idea of spending as little time as possible in requirements, and get on with design and coding. However, there is still the fact that, compared to the requirements phase, errors in design cost 3-6 times more to fix, while the cost during implementation can be up to 10 times more (Table 1). Dr. Brooks suggestion is not to spend less time in the requirements phase, but to extend the requirements phase to include some design in the form of a throw away prototype [7, 8]. In this approach a prototype, and the valuable information learned from interacting with it, is used to clearly define the requirements.

To some the requirements phase primarily means writing, reviewing, and maintaining a document. The fact is much research has been conducted in ways to extract the most benefit from requirements engineering, with the least amount of manual labor. Various tools now exist that capture requirements in the form of a modeling language, do some degree of validation, and then generate code automatically [33]. Such tools enable the engineer to spend more time understanding the problem domain, and perhaps even eliminate the ambiguity-prone natural language document all together.

Some of this research includes more ways to use the classic Artificial Intelligence (AI) technique of knowledge representation and reasoning. This means that requirements engineering has been trending toward the use of requirements modeling and the automatic analysis of the resulting models

via computer. Other areas of research include problems that can be exposed and addressed while in the requirements phase.

We will refer to these potential problem areas as *areas of concern*. And while a lot of this research is conducted with the IT domain in mind, most of it can be readily applied to embedded systems. In this paper we will look at the activities that make up requirements engineering, and the characteristics that define an embedded system. Additionally, we will look at the areas of concern that are being researched and addressed in the requirements phase, namely: Operational Context, Vacuity (pre and post-conditions), Stakeholder Viewpoints, Traceability, Modeling of Temporal Characteristics, and Managing of Changing Requirements. Lastly, we will see how considering these areas of concern can help improve the quality of embedded systems requirements.

REQUIREMENTS ENGINEERING

We begin by defining a requirement as a capability that is necessary to solve a problem. Quite often a set of requirements is viewed as a problem statement that defines what needs to be solved [18]. For our purposes, the problem to solve is to implement, in software, a system that performs a set of tasks desired by a group of stakeholders. The resulting embedded system is considered the solution to the problem statement. In fact, in the IT world, software is sometimes referred to as a solution. To many, the term requirements brings to mind a document such as the IEEE Std 830 - 1998, thus, one can also view a set of requirements as a specification contract between stakeholders and a software engineer.

Typically a distinction between functional requirements (FR) and non-functional requirements (NFR) is made. A functional requirement specifies an action, or behavior, that must be performed [17, 18]. In the case of embedded systems, the behavior is often in response to input stimulus such as a user input and/or sensor datum. A non-functional requirement specifies a system property [17, 18]. This property can be an implementation constraint, an environmental constraint, or a performance constraint. The discipline that manages the acquisition, assessment, documentation and validation of requirements is called Requirements Engineering. A person practicing requirements engineering is a Requirements Engineer (RE). In industry the software engineer often wears the hat of RE, a responsibility that more often than not is performed with minimal formal training in requirements engineering.

Requirements Engineering Activities

Requirements engineering typically involves four major activities: elicitation, analysis, documentation, and validation. Analysis and documentation can occur in either order or even concurrently, whereas elicitation always occurs first and validation last. Note also that in methodologies that fall under

Agile or with the Spiral method, the four activities can be repeated in an iterative manner [22, 23]. We now briefly explain the four activities.

1. Elicitation involves the gathering, or discovering, of requirements, typically from the stakeholders who are seeking a solution. Elicitation can be performed in various ways, ranging from one on one interviews to group sessions, where all the stakeholders are gathered together to discuss and gather the requirements. This phase is often deceptively hard to perform, because of human cognitive factors [24].

2. Requirements analysis is sometimes treated as a negotiation phase. During this phase the Requirements Engineer (RE) and stakeholders review the captured requirements to arrive at a common understanding, eliminate ambiguities, and establish the feasibility of the requirements. Others may treat the analysis phase as precursor to the design phase. Some may not even have an analysis phase and process directly from Elicitation to design.

3. Documentation of the requirements can occur in either a natural language or in the form of a requirements model. Natural language documentation is more prevalent due to the expressiveness of a natural language such as English, and because it readily allows for non-technical stakeholders to participate in the requirements process [25].

4. A fourth step which tends to get minimal attention is the validation of requirements. Most organizations will, at least, conduct inspections of the requirements document and/or models [31, 32, 34]. Often the inspections amount to the document being skimmed over, sometimes prompted by the attitude: “Why spend a lot of time reviewing a document that is going change, or become obsolete?” In the case of safety critical requirements, a more rigorous validation will often occur using formal verification techniques. However, these techniques require personnel trained in formal methods and tend to exclude non-technical stakeholders that are domain experts.

Of the four activities, the two most consistently implemented in industry are elicitation and documentation, mainly because of their absolute necessity. Figure 1 shows how requirements engineering typically fits into the software development cycle. The solid lines show the normal work flow, from requirements engineering activities to design, implementation, and testing. Many do not view the first step as requirements engineering, but rather merely the first phase in the development process.

However, calling the first phase an engineering endeavor is not overstating the importance of the phase. Recall that the requirements phase is the only phase that operates in the problem domain. Furthermore, the transition from requirements to design is where the first act of formalization occurs [28, 29]. Every phase from design to the end product deals with the creation of a formal solution to the informal problem defined in the requirements phase. One rule of thumb that many follow when compiling requirements is that

a set of requirements should describe *what* the system will do, and not *how* it will do it.

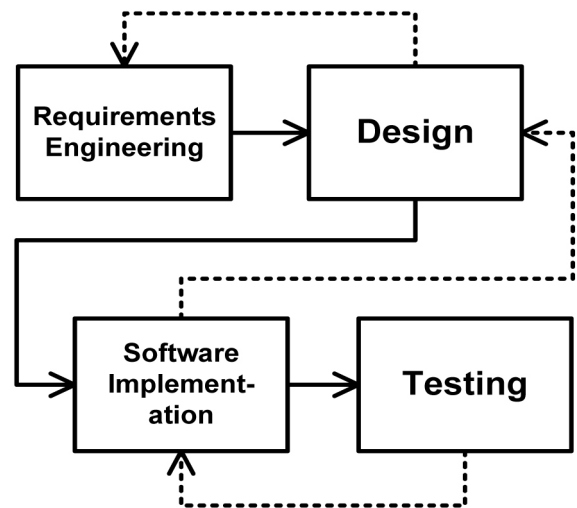


Figure 1. Waterfall type software development process with some iteration paths (dotted lines)

The question of how is answered during the design phase. Figure 1 depicts a waterfall like process, which is rarely followed in the stricter sense. However more commonly used processes such as Spiral, Agile, and Prototyping have a waterfall aspect to them, in that similar phases are implemented in a similar ordering. Strict waterfall processes are to be avoided because in the real world one does not totally exit one phase and enter another, never to revisit a prior phase. Typically, prior phases are revisited as the system-to-be is developed and new information is learned [8]. To account for this revisiting, the waterfall of Figure 1 has been modified with feedback loops (the dotted lines) that provide iteration paths between every two phases. In reality, the iteration paths can occur between any two given phases. Regardless of the development methodology used, the requirements engineering phase is typically treated as the initial phase. This is because it is a generally accepted practice to gather as many functional and non-functional requirements as possible before the design process can begin. Of course, as new requirements are learned the requirements phase should be revisited to a lesser degree; in practice, this is not always the case.

When requirement changes occur, and the requirements document is updated, the changes will tend to have a ripple effect as they propagate downstream through the development phases. This contributes to a lot of the problems that arise in a typical project, mainly because the propagation involves human effort. There is another approach to software development which automates some of this ripple effect, and is becoming more commonly utilized in industry. Figure 2 shows a general representation of what is commonly called Model Driven Development (MDD). In this methodology the requirements are converted into a design model from which software and a set of test cases are then automatically

generated. Note that the iteration feedback primarily exists between the requirements phase and the design model.

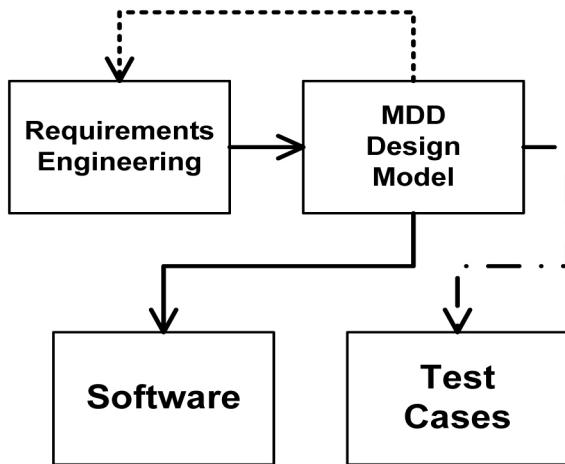


Figure 2. Model Driven Development process

Also note that the path from design model to test cases is also dashed because not all MDD systems generate test cases. One advantage of the MDD approach is that it facilitates the managing of changing requirements. As the requirements change, a new model is produced and new software and corresponding test cases are generated in turnkey fashion. Thus, the changes in requirements only need to propagate to the design model, since the software and test cases are automatically generated in their entirety, and the prior revisions are discarded. One example of the MDD approach is Rational Rose Real Time, which uses a finite state machine modeling approach and automatically generates C code.

Another commonly used MDD system is one found in Matlab/Simulink. Regardless of whether one is using a classic multi-phase methodology (Figure 1) or a model driven development approach (Figure 2), there is always the need to convert a set of requirements, typically written in a natural language to a design. This amounts to converting an informal representation of the requirements into a formal representation [29]. The formalizing of Natural Language (NL) to a design model is still primarily done in a manual ad-hoc manner. Natural Language Processing has still not advanced to a point where a flawless automatic formalization is possible without human review. This is due in part to the ambiguities found in natural languages [25]. This means that the creation of a design model from a set of NL requirements is still performed by an engineer trained in such formalizations. Many engineers are not specifically trained at such a task, but still perform it using experience and a working knowledge of such modeling languages as Unified Modeling Language (UML). The goal in formalizing a set of NL requirements is to eliminate incompleteness, inconsistency, and ambiguity [29]. These are the three primary error classifications found in NL requirements [2, 18].

One other aspect of requirements engineering that should be noted is what is sometimes referred to as organizational issues [30]. These involve factoring in the number of stakeholders that must be elicited, the maturity of the software house's requirements process, and whether the software development team is distributed or centralized. In industry, the requirements engineering process is often conducted with the aid of RE support tools that can be classified as modeling/validation tools and management tools [30]. Management tools help to manage the changes in requirements that are a normal part of the software development process.

The requirements model

The modeling of requirements has become a very useful endeavor which enables the automatic validation of requirements, and results in fewer instances of ambiguities. For our discussion, a model is an abstract representation of a system, in which a particular aspect of that system is emphasized in the representation [32]. That being the case, it is often necessary to model a complex system using more than one model; each model would then offer a different view of the system. This enables an analyst of that system to focus on one system view at a time. Models are commonly used in the design phase of a systems development, where often the UML, with its multiple modeling views, allows for the complete modeling of system, both from a structural and behavioral perspective. Models can also be used during the requirements engineering phase as an alternative (and/or compliment) to natural language representation of requirements.

Modeling a set of requirements helps convey information to other stakeholders in a non-ambiguous manner. It also allows for the automated analysis of requirements. Requirements can be modeled mathematically, descriptively, or graphically. A mathematical representation involves the use of a formal and symbolic language that can be manipulated using a set of logic-based rules. Descriptive models can include the use of scenarios, which many in industry have found to be very effective in capturing desired system behavior from non-technical stakeholders [34, 35]. Graphical representation involves the use of diagrams, the most popular in embedded system requirements being variations of a finite state machine. This is due to the state transition nature of reactive systems; reactive systems don't typically terminate, but tend to be in one or more concurrent states at any given time.

When modeling the requirements of an embedded system, the emphasis is often on modeling the behavior of the system. An embedded system typically has more than one type of behavior, each of which should be taken into account when modeling requirements. This includes the following behaviors.