Two techniques for software safety analysis

Dingding Lu

Iowa State University
Two techniques for software safety analysis

by

Dingding Lu

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science
Program of Study Committee
Robyn R. Lutz, Major Professor
Gary T. Leavens
James Davis

Iowa State University
Ames, Iowa
2003

Copyright © Dingding Lu, 2003. All rights reserved.
This is to certify that the master's thesis of

Dingding Lu

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iv</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td><strong>CHAPTER 1: FAULT CONTRIBUTION TREES FOR PRODUCT FAMILIES</strong></td>
<td>5</td>
</tr>
<tr>
<td>1. Introduction and background</td>
<td>5</td>
</tr>
<tr>
<td>2. Related work</td>
<td>8</td>
</tr>
<tr>
<td>3. Overview</td>
<td>10</td>
</tr>
<tr>
<td>4. Domain engineering</td>
<td>12</td>
</tr>
<tr>
<td>4.1 Two-dimensional view of product-family domain</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Commonality and variability analysis</td>
<td>13</td>
</tr>
<tr>
<td>4.2 The commonality/variability tree</td>
<td>16</td>
</tr>
<tr>
<td>4.3 Fault Contribution Tree Analysis (FCTA)</td>
<td>19</td>
</tr>
<tr>
<td>5. Application engineering</td>
<td>21</td>
</tr>
<tr>
<td>5.1 Reusability – prune FCT for family members</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Adaptability – construct FCTs for new family members</td>
<td>25</td>
</tr>
<tr>
<td>5.3 Evaluating efficiency</td>
<td>26</td>
</tr>
<tr>
<td><strong>CHAPTER 2: CONSTRUCTING SAFETY-RELATED SCENARIOS TO SUPPORT</strong></td>
<td>29</td>
</tr>
<tr>
<td>ARCHITECTURAL EVALUATION</td>
<td></td>
</tr>
<tr>
<td>1. Introduction and background</td>
<td>29</td>
</tr>
<tr>
<td>2. Related work</td>
<td>35</td>
</tr>
<tr>
<td>3. Overview of the approach</td>
<td>36</td>
</tr>
<tr>
<td>4. Global analysis</td>
<td>45</td>
</tr>
<tr>
<td>5. The four-view system decomposition</td>
<td>52</td>
</tr>
<tr>
<td>5.1 The definitions of the four views</td>
<td>52</td>
</tr>
<tr>
<td>5.2 The activities of each view</td>
<td>54</td>
</tr>
<tr>
<td>5.2.1 Hazards derivation and prioritizing ---- SFMEA</td>
<td>54</td>
</tr>
<tr>
<td>5.2.2 Hazard analysis and scenario derivation ---- SFTA</td>
<td>57</td>
</tr>
<tr>
<td>6. Discussion of results</td>
<td>63</td>
</tr>
<tr>
<td>7. Sample Application: the insulin delivery system [25]</td>
<td>65</td>
</tr>
<tr>
<td>7.1 Perform the global analysis</td>
<td>65</td>
</tr>
<tr>
<td>7.2 Perform the four-view’s system decomposition</td>
<td>66</td>
</tr>
<tr>
<td>7.2.1 The conceptual view of the insulin delivery system</td>
<td>66</td>
</tr>
<tr>
<td>7.2.2 The module view of the insulin delivery system</td>
<td>70</td>
</tr>
<tr>
<td>CONCLUSION</td>
<td>79</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>80</td>
</tr>
</tbody>
</table>
ABSTRACT

Currently many safety-critical systems are being built. Safety-critical systems are those software systems where a single failure or hazard may cause catastrophic consequences. Therefore, safety is a property that must be satisfied for safety-critical systems. This research develops techniques to address two areas of software safety analysis in which structured methodologies have been lacking. The first contribution of the paper is to define a top-down, tree-based analysis technique, the Fault Contribution Tree Analysis (FCTA), that operates on the results of a product-family domain analysis. This paper then describes a method by which the FCTA of a product family can serve as a reusable asset in the building of new members of the family. Specifically, we describe both the construction of the fault contribution tree for a product family (domain engineering) and the reuse of the appropriately pruned fault contribution tree for the analysis of a new member of the product family (application engineering). The second contribution of the paper is to develop an analysis process which combines the different perspectives of system decomposition with hazard analysis methods to identify the safety-related scenarios. The derived safety-related scenarios are the detailed instantiations of system safety requirements that serve as input to future software architectural evaluation. The paper illustrates the two techniques with examples from applications to two product families in Chapter One and to a safety-critical system in Chapter Two.
INTRODUCTION

Currently many safety-critical systems are being built. Safety-critical systems are those software systems where a single failure or hazard may cause catastrophic consequences. Therefore, safety is a property that must be satisfied for safety-critical systems. This research develops techniques to address two areas of software safety analysis in which structured methodologies have been lacking. The work reported here has produced two new techniques, one to improve the safety analysis of critical product families and one to construct safety-related scenarios to support software architectural evaluation.

Software product families are being widely developed in the software industry today. A product family is a set of systems sharing a common, managed set of features satisfying a particular market segment or mission [1]. Each family member is an individual system, or is also a sub-family itself.

The main concept of product famil}~ is that all family members share a set of common features called commonalities, and they are distinguished from each other by a few key differences called variabilities [2,3]. For example, in a floating weather station product family [3], all weather buoys are equipped with some sensors. However, the type and the number of sensors used by each buoy may vary. Thus, sensor equipment is the commonality; the different sensor type and quantity are the variabilities.

Critical product families are those product families developed and deployed in domains such as aviation, power control systems, spacecraft, and medicine. Safety is the key property needing to be satisfied in critical product families because a single hazard or failure may yield loss of mission or even human lives.

Reusability is a key benefit of product families. The product family systems can be built from a common set of core assets [2,3] ---- the commonalities. Thus, a substantial cost saving and reduced time to market can be achieved.

By introducing the concept of software product family, reuse can be achieved in almost every development stage. Although reuse across product family domain brings great advantages and benefits, it involves significant safety issues. For example, inconsistency may occur between the reused requirement and the new requirement introduced by a new family member. Particularly for critical product families, the system safety must be analyzed and satisfied.
The current existing problem is that critical product families are being widely developed without adequate use of safety-analysis mechanisms.

In Chapter One of the works presented here, we define a new technique called Fault Contribution Tree Analysis (FCTA) to reason about the safety issue of product family software. The FCTA is a new kind of top-down, tree-based analysis technique which is similar to Software Fault Tree Analysis (SFTA). FCTA takes a potential hazard as root node, works backward to the product family requirement domain and finds out a possible combination of features which may contribute to the occurrence of the root hazard. By doing so, FCTA has pointed out those features which may involve hazards. Thus the safety of product family software can be satisfied by taking care of those hazardous features.

The contribution of Chapter One is to define the FCTA technique and address how it is designed to analyze the safety of the product family software. Also we explore how and to what extent FCTA can serve as a reusable product-family asset. The FCTA can be reused during development of either an existing family member or a new family member. In this way, we are able to utilize the outstanding property of the product family development, that is, the core assets can be reused, while at the same time the safety of the system will still be satisfied.

Though safety is the key property of the critical product families, not all aspects of the system are related to safety. How to identify and refine those safety-related requirements becomes critical when a product family containing many systems is considered. By identifying those safety-related requirements, it can guide the engineers to put emphasis on the most significant parts of system development and allocate development resources efficiently.

The safety-related requirements must be satisfied during the software development. As an important aspect of software development, software architecture (the description of elements from which systems are built, interactions among those elements, patterns that guide their composition and constraints on these patterns [4]) will influence some qualities of the system such as safety and reliability. Thus, the available software architectural styles (the high-level patterns) must be evaluated against the safety requirements so that a good selection can be made to build the system.

If more than one software architectural styles can be chosen to build the system, an architectural evaluation method needs to be used to tradeoff all the advantages and disadvantages among those architectural approaches. Based on some predefined screening criteria, the most suitable approach will be selected.
The Architecture Tradeoff Analysis Method (ATAM) [5] is an architectural evaluation method which uses scenarios to evaluate and tradeoff among different architectural styles. ATAM is a good starting point for software architectural evaluation and has produced promising results when applied in industry. However, the input scenarios came from brainstorming of experts rather than from a systematic approach. This limitation of ATAM causes problems such as lack of coverage, guidance, detail, and traceability.

Motivated by the above problems, we define an improved method in Chapter Two. The method will be executed before the architectural evaluation and provide an input to the architectural evaluation.

The approach has two bases. One is the complete view of the system. The complete views of a system refer to the four different architectural views developed by Hofmeister, Nord and Soni [6]. The four views (conceptual view, module view, code view and execution view) describe the different levels of system structure from four different perspectives of system development. By decomposing the system into these four views, relative completeness and detail can be achieved.

The second base of the approach is the hazard analysis methods such as Software Failure Mode and Effect Analysis (SFMEA) [7] and Software Fault Tree Analysis (SFTA) [8,9,10,11]. Those hazard analysis methods serve to identify the safety-related requirements and refine them into scenarios. Thus, the derived scenarios are instantiations of the safety-related requirements not the casual results from the brainstorming of experts.

The contributions of the work presented in Chapter Two are as follows:

First, these results provide a systematic way to decompose the system. The four system views deal with different engineering concerns from four perspectives. Separated into four views, the different engineering concerns can be handled independently within each view. By deriving safety-related requirements for each of the four views, a relatively complete coverage can be achieved.

Second, this work combines scenarios derivation with hazard analysis. The hazard analysis guides engineers to look at the significant parts of system during safety-related requirements derivation. It prevents the engineers from exploring in a wrong direction and thus achieves efficiency both in development and in resource allocation. By deriving scenarios directly from the results of hazard analysis, the safety-related requirements are instantiated into detailed system behaviors. Thus, the traceability from the safety-related requirements and the detailed scenarios can be achieved.

The remainder of the paper is organized as follows:
Chapter One presents the Fault Contribution Trees for product families. It defines the Fault Contribution Tree Analysis (FCTA) method and explores its use in safety analysis for product families. It provides a feasible way to reuse FCTA during development and evolution of the product family. At the end of this chapter, we discuss several challenges to this approach, including handling of evolution of the product family subfamilies, and suggest partial solutions to these issues as well as directions for future work.

Chapter Two presents a method for constructing safety-related scenarios to support architectural evaluation. In this chapter, we define a process to derive undesirable hazards and misuse scenarios [26] which may compromise the safety of the systems. We describe the two bases on which the approach is built. The four system views decompose systems based on four different perspectives of engineering concerns. The hazard analysis methods provide a systematic way to derive scenarios which are safety-related. Finally, the software architectural evaluation process (which takes the derived scenarios as input) is discussed and its feasibility is considered.

The final section of the thesis summarizes the contributions and the results of the two chapters and suggests some directions of future work.
CHAPTER 1: FAULT CONTRIBUTION TREES FOR PRODUCT FAMILIES

1. Introduction and background

This chapter investigates the extension of a safety analysis technique similar to Software Fault Tree Analysis (SFTA) to product families. The chapter has been published in [12]. SFTA is an important analysis technique that has been used successfully for a number of years and in a variety of critical applications to verify requirements and design compliance with robustness and fault-tolerance standards. Historically, fault tree analysis has been applied mainly to hardware systems [13], but good results have been obtained in recent years by applying the technique to software systems as well [8,9,10,11].

SFTA uses Boolean logic to break down an undesirable event or situation (the root node) into the preconditions that could lead to it. SFTA is thus a top-down method that allows the analyst to explore backward from the root node to the possible combinations of events or conditions that could lead to the root node's occurrence. When SFTA is used for safety analysis, the root node is a hazard that we wish to avoid.

We here define a new kind of top-down, tree-based analysis technique, Fault Contribution Tree Analysis (FCTA) that is similar to SFTA but operates on the results of a product-family domain analysis rather than on the design logic of a particular system. The FCT shares with a software fault tree (SFT) the use of Boolean logic to describe the combinations of nodes that may contribute to a particular root hazard or failure. The FCT is different from a SFT in that:

- The nodes of the FCT are features (commonalities or variabilities) rather than events or conditions as in the SFT.

- Lower nodes in the FCT are refinements of higher nodes, whereas in the SFT lower nodes may also convey a notion of time (with lower nodes preceding higher nodes)

- The logical expression represented by the FCT states that if some combination of features is selected or has certain values, then the root node is a hazard that needs to be investigated for this system. That is, the hazard cannot be ruled out at this point so requires further investigation (e.g., via a SFTA once the design exists). This contrasts
with the SFT which states that if some combination of events or conditions occurs, then the root node will occur.

The FCT can thus serve as a "preprocessor" to the construction of a SFTA for a system in the product family. The FCT identifies the subset of features, first for the product family and then for the particular system, that may be able to contribute to the hazard being investigated.

The concern that motivates the work described here is that critical product lines are currently being developed and deployed in domains such as aviation, power control systems, spacecraft, and medicine without adequate use of safety-analysis techniques. We here define a product line as a set of systems sharing a common, managed set of features satisfying a particular market segment or mission [1]. Most product lines are also product families in that the systems are built from a common set of core assets [2, 3]. These assets routinely include shared requirements, architecture, code, and test cases, as well as shared analyses such as commonality analysis, dependency analysis, architectural analysis, and performance analysis. The existence of these reusable assets can provide reduced development costs, faster time to market for new systems, and improved identification and management of risk factors for a company using a product-line approach.

As more companies adopt a product-line development method, more safety-critical product families are being built. However, little work has been done to extend existing safety-analysis techniques to handle product families [10, 14]. One reason for this gap between the number of safety-critical product families and the lack of safety-analysis techniques to support them is that any effort to reuse safety analyses must be very carefully constrained. We do not currently know how to safely reuse safety analyses. Safety is a property of a particular system operating in a particular context, not a property of a set of similar systems. However, we speculate that some degree of reuse of safety analyses is both possible and beneficial within the context of product families. In part, this is because the alternative to date has been no SFTA at all for these systems.

The contribution of this chapter is to explore how and to what extent FCTA can serve as a reusable product-family asset. Our approach involves two phases, mirroring the distinction between the domain-engineering phase in which the product family is defined and the application-engineering phase in which product-family systems are built [2, 3]. We first define a method by which a fault contribution tree can be initially constructed for a product family during the domain engineering phase and then describe how that fault contribution tree can be reused for FCTA as individual members of the product family are built. The two-phased approach thus provides (1) an initial safety
analysis of the entire space of anticipated systems in the product family and (2) the subsequent safety analysis of each new system as relevant portions of the baseline FCTA are reused.

The primary challenges that must be addressed in the domain engineering phase of the FCTA are how to structure the specifications to support scalability (e.g., many features) and how to efficiently manage the notion of subfamilies within the baseline product family. The primary challenges that must be addressed in the application engineering phase of the FCTA are appropriate pruning techniques for the baseline fault contribution tree (e.g., when optional features are excluded) and how to handle the inevitable evolution of the product family (e.g., as unanticipated features or options are added).

The rest of the chapter is organized as follows. Section 2 describes related work. Section 3 provides an overview of the approach. Section 4 describes the commonality and variability analysis required to construct a FCTA and the extension of FCTA to the domain engineering phase. Section 5 describes the reuse of the FTCA in the application-engineering phase together with techniques for pruning and evolution and presents some concluding remarks.
2. Related work

The research work reported here builds on two bodies of related work: safety analysis and product families.

In the area of safety analysis, recent work by Dugan, Sullivan, and Coppit [15]; Hansen, Ravn and Stavridou [16]; Leveson [9]; and Lutz and Woodhouse [11] on fault tree analysis and software fault tree analysis has demonstrated both the continuing use by developers of these top-down analysis techniques and their power to identify and remove unsafe or failed conditions at an early stage of development.

In the area of product families, our work is based on the commonality analysis in the FAST method of Ardis and Weiss [2] and Weiss and Lai [3], whose Floating Weather Station example we use to illustrate our approach. Like them, and like Thompson and Heimdahl [17], we find it useful to divide our domain into a hardware and a behavioral dimension in order to reduce complexity and decouple changes in one dimension from changes in the other. We adopt Thompson and Heimdahl’s notion of a hierarchical product family (especially the notion that subfamilies may contain additional commonalities as well as variabilities) to describe evolving product families. We use their evolving Mobile Robot product family to illustrate our FTCA technique.

Our commonality/variability graph is somewhat similar to the representation of a decision model for a product family by a variability tree in Lam [18]. By using a graph, we support partial ordering of decisions and avoid the imposition of a total ordering of decisions among levels as required by a tree. Lam suggests as a guiding factor to “place more ‘significant’ variability higher up the variability tree.” Our use of refinement as the structuring mechanism for the graph implements this suggestion.

Gomaa’s EDLC (Evolutionary Domain Life Cycle) [19] also uses refinement to structure the representation of variabilities, but at an implementation rather than a requirements level. He describes an aggregation hierarchy to model optional object types which allows refinement of those objects representing Boolean variabilities.

FORM (Feature-Oriented Reuse Method) [20] is another representation of a decision model that provides a hierarchical notion of refinement, there based on composition and generalization relationships. Like our Fault-Contribution Tree, FORM represents commonalities and variabilities in the same structure (there, an AND/OR graph). Unlike our FCT, the purpose of the representation is
not to identify combinations of features relevant to safety or failure analysis, but to trace relationships between features in the decision model.
3. Overview

Figure 1 below provides an overview of the approach. The left-hand side of the figure describes the activities involved in specifying the requirements for a product family and in performing the safety analysis of the product family. These activities to define a product family and produce its reusable assets are commonly called “domain engineering” [3]. The activities in the two sides of the figure are described in Section 4. The horizontal arrows between the right-hand side of the figure describe the potential reuse of the member of the product family is built. These are the activities involved for the new family member and in performing the safety analysis for process of developing new members of the product family is commonly called.” This process is described in Section 5. We will use two product-families, the Floating Weather Station (FWS) product family [2, 3] and the family [17], to illustrate our approach throughout the chapter.

We encounter when trying to apply Fault Contribution Tree Analysis how to manage size and scale. This problem motivates us to first focus on the family domain to support scalability. The boundary of the domain is defined as the area which are shared by all family members. The family members are defined by their variabilities. A two-dimensional view of the domain allows us to enforce a tree structure for the specification model. In the product family domain, we can start to apply FCTA to it. In the Domain of the two parallel processes. One is the specification of requirements for the hazards and safety analysis. These two processes can be assigned to do their work individually and simultaneously. In this chapter we do not do the hazards of a specific product family since this is well documented under the assumption that hazard identification has been completed.
Specify the requirement of product family

Choose the combination set of commonality & variability

Specify the requirement for new family member

Prune tree for family member

Safety analysis for product family

Domain Engineering

Safety analysis for new family member

Application Engineering

In the application engineering phase, there are again two parallel processes: the specification of the new system being built and the safety analysis of this new system. Because each system is unique, even if it is a member of a product family, methods for handling new commonalities and variabilities, methods for delimiting the safe reuse of the FCTA, and methods for appropriately pruning the product-family fault contribution tree for the new member are required. The hazards previously found in the domain engineering phase as well as the FCTA previously constructed for the product family are thus used as a baseline for the safety analysis of individual members.

The contributions made to safety analysis for software product families by this two-phased approach are: (1) to provide a structured safety-analysis approach to domain and application engineers, (2) to pursue the safety-analysis goal from a practical perspective that adapts to evolving product families, and (3) to support the reuse of a product-family safety analysis for improved coverage of individual systems in the product family.
4. Domain engineering

The domain is first defined as the union of two dimensions: hardware and behavior [3]. A large part of the initial effort involves the commonality and variability analysis of the envisioned product family, the grouping of requirements into dimensions, and the organization of each dimension based on a hierarchical perspective. After domain experts have performed a preliminary hazard analysis, the domain engineers can begin the Fault Contribution Tree Analysis (FCTA). Each identified hazard becomes the root node of two fault contribution trees, one for each of the hardware and behavioral dimensions.

4.1 Two-dimensional view of product-family domain

Because most product families are embedded systems, Weiss and Lai [3] have introduced the two-dimensional view of a product family. The hardware dimension consists of features that are related to the hardware components of the system. An example of a requirement common to all systems in their Floating Weather Station (FWS) family from the hardware dimension is, “The FWS is equipped with one or more sensors that monitor wind speed.” The behavioral dimension, on the other hand, represents the functionalities provided by the software. An example of a commonality requirement from the behavioral dimension is, “At fixed intervals, the FWS transmits messages containing an approximation of the current wind speed at its location.” The union of the two dimensions covers the original domain.

We can distinguish most characteristics of product families as belonging to either the hardware or the behavioral dimension. However, some features may have properties of both hardware and behavior. An example is the requirement, “Each sensor on board an FWS has a way to indicate its reliability.” Features such as these, which have characteristics of both dimensions, we call “boundary features (Figure 2).” Since the goal is to ensure that the union of the two dimensions will cover the whole product-family domain, it is feasible to regard the boundary features as belonging to either of the dimensions.
There are several advantages to adopting the two-dimensional view for product families. First, due to the complicated nature of many real-world product families, it is hard to organize the entire domain in a hierarchical fashion. By decomposing the domain into two dimensions, we can more readily achieve an intuitively reasonable hierarchy within each dimension. Second, the two-dimensional view reduces the depth of subsequent decomposition results as shown in Figure 3. The horizontal direction describes the two-dimensional division into hardware and behavioral characteristics. The vertical direction hierarchically organizes the commonalities and variabilities within each dimension.

3.2 Commonality and variability analysis

Section 4.1 described the horizontal decomposition of the product-family domain into two dimensions. This section describes the vertical or hierarchical analysis of the commonalities and variabilities within each dimension as shown in Figure 4. Because our primary interest in this work is on the software, we concentrate here on the structure of the behavioral dimension. However, the techniques described also work for the hardware dimension.
To perform the commonality and variability analysis of the product family, we follow the procedure provided in [2, 3]. To enhance the hierarchical structure within each dimension, we extend the analysis by introducing another rule to enforce the vertical hierarchy. By this rule, the modeling of the commonality and variability requirements proceeds from the most abstract to the most detailed. That is, the analysis begins with the general features of the product family and then decomposes these into sub-commonalities and sub-variabilities. The commonality and variability analysis thus yields a structured model, in particular, a directed acyclic graph representing the partial order "is a refinement of," with each level of commonalities and variabilities providing a more detailed description of the level above it.

The most common type of variability in any product family describes whether or not family members have a specific feature. An example of this Boolean type from the Mobile Robot product family is, "A robot may or may not be equipped with a camera" [17]. Any subsequent vertical decomposition of a node representing a Boolean variability will only have meaning for systems in the product family that have this feature. If a particular system does not have this feature, it can not have any subfeatures that may be associated with it. For example, a system that does not have a camera will not have the subvariability, "The camera may be digital or not."

At the leaf-node level, each non-Boolean variability is annotated with the range of values that the variability can take. That is, each leaf node corresponds to a parameter of variability in Weiss and Lai's terms. The labeling of leaf nodes with possible values enables the Fault Contribution Tree Analysis to be constructed directly from the information in the commonality and variability model.

A special case that must be considered in performing commonality and variability analyses is the subfamily. Thompson and Heimdahl have discussed the usefulness of the notion of subfamilies, particularly as product families evolve, in [17]. They give as an example the Mobile Robot family which consists of four distinct subfamilies. A family member can either be an individual product or a set of products (sub-family) that all share a set of additional commonalities and differ from one another within this subfamily by possibly additional variabilities. Hence, a sub-family can introduce additional commonalities and variabilities, whereas a new individual product usually only introduces additional variabilities.
CB: At fixed intervals, the FWS transmits messages containing an approximation of the current wind speed.

CB2: The wind speed value transmitted is calculated as a weighted average of the sensor readings, calculated over several readings for each sensor.

VB: The formula used for computing wind speed from the sensor readings can vary.

VB2: The types of messages that an FWS sends can vary according to both content and format [short-mesg, long-mesg].

VB3: The transmission period of messages from the FWS can vary.

Figure 4. A portion of commonality and variability analysis for FWS in behavior dimension

To distinguish the number of subfamilies within the product family being discussed, we prefix the subfamily designation with a cardinal number. Thus, a family consisting of two subfamilies is referred to as a 2-subfamily product family. Every member of an n-subfamily product family is itself a product family (subfamily) that includes a set of individual products. The structure of an n-subfamily product family is hierarchical with the parent subfamilies inherited by every child subfamily. Every child subfamily has all the commonalities and variabilities of its parent and can also introduce additional commonalities and variabilities of its own. Figure 5 depicts the hierarchical relationship among subfamilies in a 9-subfamily product family.

Figure 5. The hierarchical relationship among sub-families
Within each subfamily we use the commonality and variability analysis method introduced at the beginning of this section. The inheritance structure among the commonalities and variabilities identifies the hierarchical relationship among the subfamilies. Figure 6 shows the inheritance relationship among the four subfamilies of the Mobile Robot product family [17].

![Figure 6. Sub-families of the Mobile Robot product family](image)

The approach to commonality and variability analysis described here provides three benefits:

1. By decomposing the product family domain step by step, it is more possible for the domain engineers to understand the entire system and the relationship among family members at the early stage of development.

2. The resulting top-down structure for the representation of commonalities and variabilities in each dimension facilitates grouping them into trees in the next section. The high degree of consistency between the representation of the commonality analysis results, the resulting commonality/variability tree, and the subsequent fault contribution tree provides better understanding for the users and encourages cross-checks of the accuracy of representations.

3. The process by which the analysis moves from abstract to detailed features matches the top-down process by which a FCTA is constructed. The safety analysis thus becomes an extension and ideally a parallel development to the domain analysis.

### 4.2 The commonality/variability tree

The graph produced in the previous step can be readily converted to a tree in two simple steps. As an example, Figure 7 shows the commonality/variability tree produced from the commonality and variability graph of Figure 4.
1. Convert the graph to a tree.

The enforcement of a hierarchical approach in the commonality and variability analysis will not necessarily have yielded a tree structure. Because some low-level commonalities or variabilities may be shared by several upper-level commonalities or variabilities, it may instead be a directed acyclic graph. Figure 8 shows an example where commonality or variability D is shared by B and C.

In order to convert the graph into a tree, we duplicate the shared commonality or variability, as shown in Figure 9.
The duplication introduces a new problem. When the shared commonality or variability appears at a relatively high level, the duplication may be extensive due to the need to copy the entire subtree having the shared commonality or variability as its root node. This same sort of duplication also occurs in fault contribution trees, where it is either treated as an acceptable overhead or the subtree is spun off as a reusable component with multiple references to it. Either solution is also feasible here.

2. **Select/create root node.**

In many cases, the commonality analysis identifies a single, top-level commonality that identifies the product family. For example, for the FWS, the top-level commonality might be "Provides weather information". In the case where the hierarchical graph produced in 3.2 has more than one top-level node, we add a single node above the existing top-level nodes and designate it as the root node.

Grouping the commonalities and variabilities together in a tree has two key advantages over the more-common consideration of commonalities and variabilities separately:

First, the identification of the relationships among commonalities and variabilities during this construction is very valuable to the safety analysis. This representation of possible relationships aids in tracing the possible contributions of the many commonalities and variabilities to potential failures. Second, by detailing the commonalities and variabilities level-by-level using a tree representation, we can refine the definition of the set of family members until it reaches the most basic statements of the requirements. This lowest level of the commonality and variability descriptions describes the range of possible values that each variability can assume.

Taking Figure 4 as an example, let us suppose that there is a family member, M, defined at its highest level of abstraction (level 1) by a single commonality, CB1, and a single variability, VB. Thus, M= {CB1, VB}. From the CB1 tree, we see that we can further refine M to describe it with additional detail (level 2) as M= {CB2, VB2, VB3, VB}. Finally, we can refine M to describe all the leaf nodes (level 3) as M= {VB2, VB1, VB3, VB}. This refinement to the lowest level is very helpful in the subsequent safety analysis. When the product-family FCTA identifies a leaf node that is in the cut set for the hazard of interest, we can then identify all the family members vulnerable to that hazard by checking which members contain that leaf node in their definition sets. Thus, if during FCTA we determine that the leaf node VB1 can contribute to the occurrence of the hazard
represented in the root node, we can catch those family members containing VB1 that are vulnerable in this way to the hazard.

### 4.3 Fault Contribution Tree Analysis (FCTA)

We here summarize the process of performing a Fault Contribution Tree Analysis (FCTA) for a product family and illustrate the process by means of a portion of the resulting FCTA for the FWS product family (Figure 10).

Taking a hazard as the root node and using the commonality/variability tree previously constructed for the product family as a guide, apply FCTA to each of the two dimensions (hardware and behavior). This results in two fault contribution trees for each root hazard. In this way, we are able to provide coverage of the domain.

![Diagram](image)

**Figure 10. FCTA for FWS product family in the behavioral dimension**

Note that not all commonalities or variabilities will contribute to the occurrence of the top hazard. It is, however, essential to retain all these intermediate commonalities or variabilities as dummy nodes in the fault contribution tree, even though they cannot cause the top hazard to occur. To explain this, let us suppose that the intermediate node CB2 in Figure 10 cannot be involved in the
occurrence of a specific hazard in any member of the product family. However, this does not preclude the child node of CB2, i.e., VB1, from being involved in the occurrence of that hazard in one or more members of the product family. The intermediate nodes that cannot contribute to the hazard must thus be retained as dummy nodes to facilitate the subsequent pruning of the fault contribution tree for specific family members. However, since leaf nodes have no descendents, they are omitted from the fault contribution tree unless they can contribute to the occurrence of the hazard.

Where subfamilies exist, it may make sense to perform a FCTA of each subfamily rather than of the entire product family. This allows significant reuse since for example we can reuse portions of the fault contribution trees which belong to the topmost subfamily when we are constructing fault contribution trees for the second-level subfamilies. This allows the analysts to concentrate on the additional features of each subfamily. Taking the Mobile Robot product family as an example, we apply FCTA on the topmost subfamily (LegoBot) first. When we need to construct fault contribution trees for the second level subfamily (Pioneer robots), we can reuse the fault contribution trees constructed for LegoBot and expand the fault contribution trees to include the additional commonalities and variabilities of the Pioneer robots subfamily.
5. Application engineering

In the application engineering phase, new members of the product family are built based on the assets produced during the domain engineering phase. These assets include the commonality and variability analysis documented in the product family's commonality/variability trees, and the safety analysis documented in the product family's fault contribution trees. Referring back to Figure 1, application engineering activities include the derivation of the new system's requirements from the baselined family requirements and the derivation of the new system's FCTA from the baselined family FCTA. This section describes an algorithm for pruning the family fault contribution tree for a specific family member in order to produce a reduced fault contribution tree customized to the particular system being built. It also illustrates the reusability and adaptability of this approach and discusses the efficiency of the method.

5.1 Reusability – prune FCT for family members

One of the motivations for applying FCTA to the entire domain of product family in domain engineering phase is to reuse the fault contribution trees later on, rather than constructing fault contribution trees for each family member from the very beginning. There are two preconditions that are needed. First, in the application engineering phase, the hazards needing to be analyzed will be the same as those that have been identified in the domain engineering phase. Second, in the FCTA step in domain engineering, we have kept all the intermediate commonalities and variabilities which don’t contribute to the occurrence of the hazard as dummy nodes. Because the family members of product family will be defined by the commonalities and variabilities which they have, we are thus able to directly map the definitions of family members into the nodes of fault contribution trees.

There are two steps in the prune-and-reduce fault contribution tree approach:

1. **Represent an existing family member by a set of commonalities and variabilities.**

   After the commonality and variability analysis step in domain engineering phase, we have decomposed the entire domain into statements of commonalities and variabilities. Thus we can represent an existing family member by the commonalities and variabilities which it has. For example, in section 4.3, we represent family member $M$ as $\{C_B, V_B\}$
3. **Prune the subtrees from the original FCTA for an existing family member.**

We will use the following algorithm to explain the process of prune and reduce process.

**Input:** T—Fault contribution trees (i.e. a pointer to the root node of the fault contribution tree)

the node x in T : a commonality or variability statement or a Gate

\[ M = \{ C_1, C_2, \ldots C_i, V_1, V_2, \ldots V_j \} \] — a family member defined by the set of commonalities and variabilities which it has.

**Process:** Prune_Reduce (T,M)  

**begin:**
Mark the root node;

**Mark_Node(T,M)**  

{  
  **begin**  
  if number of child of T==0 then
  return;

  **else**
  for each child x of T do{
    if x M II Parent(x) has been marked &&
    Parent(x) is not the root node then
    Mark x;
    Mark_Node(x, M);
  }

**Remove_Node(T,M)**  

{  
  begin
  if number of child of T==0 then
    return;
else{
    for each child x of T do{
        if x has not been marked || (the number of child of x == 1 && x is a Gate) then remove x;
        Remove_Node(x,M);
    }
}
definition set of M). Second, remove all the nodes which have not been marked. Third, after removing the nodes, since the AND gate has only one child (VB), we reduce the FCT to T1 for M. We iterate step 2 as needed for as many levels of decomposition of a family member as desired.

In section 4.3, we discussed how the level of representation of a family member by a set of commonalities and variabilities proceeds from abstract to detailed. For example, as shown there, the family member M was described at three different levels of abstraction. First, at the highest level of abstraction, we can represent \( M = \{ C_{B1}, V_B \} \). Then we can further refine M to describe it at a mid-level of abstraction as \( M = \{ V_{B2}, C_{B2}, V_{B3}, V_B \} \). Finally, at the lowest level of abstraction, \( M = \{ V_{B2}, V_{B1}, V_{B3}, V_B \} \).

To prune and reduce the FCT for M, we first apply the algorithm to the original FCT with \( M = \{ C_{B1}, V_B \} \). Since M consists of \( C_{B1} \) and \( V_B \), only the paths from the hazard to these nodes together with their subtrees are retained. That is, no basic statement which is not contained in these retained nodes will be able to cause the hazard. This yields a reduced tree \( T_1 \) which has two sub-trees with \( C_{B1} \) and \( V_B \) as the root nodes.

If we wish to consider the family member M in greater (i.e., mid-level) detail, we look at \( M = \{ V_{B2}, C_{B2}, V_{B3}, V_B \} \). In this case we further apply the algorithm to the fault contribution tree \( T_1 \) with \( M = \{ V_{B2}, C_{B2}, V_{B3}, V_B \} \). As a result \( T_1 \) will be further pruned and only the four subtrees with \( V_{B2}, C_{B2}, V_{B3}, V_B \) as root nodes will be retained to yield the reduced tree \( T_2 \). If appropriate, we can again apply the algorithm to the fault contribution tree \( T_2 \) with \( M = \{ V_{B2}, V_{B1}, V_{B3}, V_B \} \).

The advantage of looking at M at these increasingly lower levels of detail is that the fault contribution tree can be pruned more extensively without loss of accuracy. That is, performing pruning of the fault contribution tree at the highest level of abstraction yields a relatively high-level (i.e., vague) description of the contributing causes to the hazard. Performing pruning of the tree at a lower level of detail provides a more-detailed description of the threats to safety. For example, high-level pruning of the FWS member M might provide a result indicating that a “wrong message type” can cause a hazard. Pruning at a lower-level of detail can provide more precise information regarding exactly what kind of “wrong message type” is of concern. For example, the FCT might indicate that sending a real-valued instead of an integer message can cause an overflow leading to the hazard.
5.2 Adaptability – construct FCTs for new family members

In practice, product families almost inevitably evolve over time. Most frequently, a new family member introduces some new feature(s) into the existing product family domain. The FCTA for a new family member manages the evolution of the product family in the following way.

1. Identification of new commonality and/or variability.

When a new family member needs to be developed, the first step is to identify any additional variabilities, or—if the family member is a subfamily in an n-subfamily product family—any additional variabilities and commonalities. For example, suppose that in Figure 4, a new family member introduces a new variability that states: “The number of readings used by the formula can vary.” Thus, we can expand the fourth level of the variability description in the table to include this new one as shown in Figure 12. The commonality and variability analysis method described in section 4.2 supports the evolution by allowing placement of the new commonalities and variabilities in the graph as appropriate. The expanded graph thus includes the new feature(s).

2. Merge any new commonality and/or variability into the commonality/variability trees.

The second step in constructing the fault contribution tree for the new family member is to use the conversion steps described in Section 4.3 to add the new nodes to the existing trees. Thus, the commonality/variability trees are also adaptable.

3. Construct FCT for a new family member.

The third step is to reuse the product family FCTA in assembling the FCTA for a new family member. Even if the new member introduces additional commonalities and variabilities of its own, it still shares a large set of commonalities and variabilities with the existing members of the product-family domain. Starting with the fault contribution trees produced for each hazard during the product-family FCTA, we first prune the fault contribution trees as described in 4.1. This allows us to leverage the safety analysis that has already been accomplished. The resulting fault contribution trees can then be extended, using the expanded commonality/variability tree as a guideline, to include any additional commonalities and variabilities of the new family member. In summary, we construct the fault contribution trees for a new family member by reusing and expanding the existing fault contribution trees instead of by building the tree from the beginning.
Up to this point we have assumed that no new hazards are introduced either by the evolution of the product family or by the introduction of new members in the application engineering phase. This assumption is dangerous, since any change (e.g., new nodes) can open up the possibility of new hazards. Thus, the process of hazard identification must be iterated with each new system built, at least to the point of discounting the possibility of new hazards. The iterative hazard identification process is, of course, simplified by the clear specification of how this member deviates from all previous family members. In the case that a new hazard is identified during the application-engineering phase that applies to both the existing family members and the new family member, the existing product-family FCTA will function as a library of reusable subtrees. When we need to apply FCTA to a family member to investigate a new hazard, we can refer to the existing FCTA to identify those commonalities and variabilities that may yield to the occurrence of the new hazard. If a commonality or variability contributes to the new hazard, that node’s subtree will be included in the fault contribution tree.

5.3 Evaluating efficiency

The techniques for reuse of safety analysis described here only make sense if the product family contains several members that merit safety analysis. If there are only one or two members that need to be analyzed, it is obvious that applying FCTA directly to those systems will be more efficient than applying FCTA to the whole domain and subsequently pruning fault contribution trees for the other member(s). For larger product families, we propose to measure the effort of analysis by the number of commonalities and variabilities that need to be analyzed during the FCTA process.

Let $n$ be the number of members in a product family. Let $K$ denote the sum of the number of commonalities and variabilities shared by all family members and $t_i$ denote the number of commonalities and variabilities which are unique to a family member $T_i$. Let $ET_i$ be the effort required to perform FCTA on family member $T_i$ and $E_i$ be the total effort to perform FCTA on all $n$ family members. Then,

$$E_1 = ET_1 + ET_2 + \ldots + ET_n = nK + (t_1 + t_2 + \ldots + t_n)$$

In contrast, with the approach described here, wherein shared commonalities and variabilities are only analyzed once, the total effort needed will be:

$$E_2 = K + (t_1 + t_2 + \ldots + t_n)$$
Figure 11. The process of pruning and reducing the Fault Contribution Tree
At fixed intervals, the FWS transmits messages containing an approximation of the current wind speed.

The wind speed value transmitted is calculated as a weighted average of the sensor readings, calculated over several readings for each sensor.

The formula used for computing wind speed from the sensor readings can vary.

The number of readings used by the formula can vary.

Figure 12. Adding a variability to an evolving product family

Thus, the effort saved by reuse of the safety analysis is $E_2 - E_1 = (n-1)K$. Especially when $n$ and $K$ are relatively large, the time and budget savings achievable with this approach appear to be significant.

This chapter has explored some ways to reason about the safety or robustness of critical product families. In particular, it has described a technique by which construction of fault contribution trees for product family can be guided by a structured commonality and variability analysis. The FCTA serves as a product-family asset that can be reused in the safety analysis of particular systems in the product family. The chapter describes an algorithm to prune the fault contribution tree for the new system and discusses how to manage subfamilies and evolution of the product family within this context.
CHAPTER 2: CONSTRUCTING SAFETY-RELATED SCENARIOS TO SUPPORT ARCHITECTURAL EVALUATION

1. Introduction and background

This chapter defines a technique to identify and refine the safety-related requirements for a system that may constrain the software architecture.

The purpose of the approach which we present here is to derive the input scenarios for software architectural evaluation in light of the safety requirements of safety-critical systems. For safety-critical systems, a single failure may cause catastrophic consequences. Thus, to prevent potential hazards in advance is as important as to design various recovery mechanisms. Therefore, during the development of a safety-critical system, the safety requirements must be satisfied in every design and development step of the system.

Software architecture which will influence the entire life-time of a system is an important way to guarantee that some qualities of the system such as safety, reliability, etc; have been satisfied. Particularly for a safety-critical system, the question of whether the software architecture will achieve the overall safety quality of the system needs to be answered before the system is built. Thus, the available software architectural styles need to be evaluated against the safety requirements so that a good selection can be made to build the system.

For this purpose, we will derive, refine and prioritize the safety requirements and use them as input to the software architectural evaluation. By evaluating different kinds of architectures against the input safety requirement, the evaluation approach will not only reveal the best software architecture which satisfies the safety requirement but also provide insight into how those safety requirements interact with each other. Also it is feasible for architects to take special care of those high priority safety requirements which need most attention during the development and are the most difficult to be changed after the system has been built and evolved.

As Bass, Clements and Kazman define: "The software architecture of a program or computing system is the structure of the system, which comprises software components, the externally visible properties of those components, and the relationships among them [21]." Software architectural styles such as client-server, pipe-filter, etc have been defined by Shaw and Garlan [4] to
describe the vocabulary of components and connector types, and the set of constraints (e.g., performance, safety, reliability) on how they can be combined. Definition of a software architecture is important in terms of opening a channel to communicate between different stakeholders of the software, capturing the specific system properties in the early design stage and providing architecture reuse for similar systems in the same domain.

When there are more than one architectural style that can be chosen for a system or a component of a system, architectural evaluation must be used to trade off advantages and disadvantages among architectural approaches provided by different styles and select the most suitable one for the system. Software architectural evaluation has been practiced since the early 90’s. The Architecture Tradeoff Analysis Method (ATAM) [5] is one such evaluation methods that has shown promising results when applied in industry. An example of the application of ATAM is to use this method to evaluate a war game simulation system [22].

ATAM takes place at a relatively high level of system design. The method adhere issues that involve the overall association of system capability with components, subsystems and the interconnections among them. All the issues are visualized as scenarios. A scenario is a series of system behaviors starting at a stimulus, causing a sequence of system reactions and ending at a specific system response [5]. The scenarios are derived from former experience checklists which come from the brainstorming of a group of experts and are relatively high level. Finally, ATAM derives an architectural style based on the overall evaluation of all the trade-offs of applying different architectural styles on each scenario. The resulting architectural style is composed of components and connectors as well as the relationships between them.

ATAM is a good starting point for software architectural evaluation. It gives an overview of several quality attributes (key properties such as performance, reliability and safety) that need to be satisfied in the architectural design. The quality attributes will be grouped under the root of a tree which ATAM calls a utility tree. The utility tree has “utility” as the root node and the quality attributes as the first level nodes. Each quality attribute will be further refined into scenarios which will be the lower-level nodes in the utility tree. The scenarios will be derived by brainstorming of experts and prioritized according to how much impact they will have on the architectural decisions.

The derived and prioritized scenarios are the inputs to the architectural evaluation step in ATAM. All the later architectural decisions will be evaluated against those scenarios by assessing the potential effect of a specific architectural decision on each scenario. The effect will be categorized
into four kinds (risk, sensitivity, tradeoff and non-risk) from the most critical to the least significant. Thus an architectural decision which incurs risk on many of the scenarios may be abandoned and those which accommodate most of the scenarios will be selected. For example, a scenario of the reliability attribute says: "No requests should be lost as a result of system overloads or failure". Thus, an architectural decision which distributes all the requests throughout the system may put high risk on this scenario. Therefore, the above architectural decision will be categorized as 'risk' from the perspective of reliability and may be eventually abandoned during the final architectural decision selection process.

From the ATAM process, we can see that the evaluation result completely relies on the scenarios. How complete the scenarios are will decide how accurate the evaluation will be. However, the input scenarios yield problems:

1. Lack of coverage. The scenarios are derived from brainstorming sessions by experts. The completeness of the scenarios determines the accuracy of the architectural evaluation, hence, the reliability of the selected architectural style in terms of building a specific system.

2. Lack of guidance. Because the evaluation inputs are the scenarios refined from the high level of system design by brainstorming, the resulting architecturally significant requirements are not detailed enough to guide the further software development processes. Without guidance an architect could easily pick the wrong details to explore in the further design plan and could miss the important ones.

3. Lack of detail. The high-level scenarios will result in a high-level architectural style after evaluation. Though the resulting architectural style may be able to guide the system development trend, it is too abstract to be useful in developing a system in practice. A detailed visualization of the architectural style into lower-level design (such as components and subsystems) and the code level is necessary for building a real system. This problem will be even more obvious when a large system is going to be built. Because a large system usually has very complicated structure with many subsystems and components, a high level design will be easily abandoned if it can not satisfy the special development requirements of the subsystems and components.

4. Lack of traceability. The selected high-level architectural decisions will not be mapped and refined step by step into detailed development processes so it is hard to trace the
design or development decisions made in the lower-level design stages back to the original architectural style. Especially in building a large system which may involve many requirement changes during the later development steps, the original architectural style risks abandonment if it can not accommodate the changes.

The above problems motivate us to think about an improved method which will be executed before the architectural evaluation and provide an input to the architectural evaluation with improved completeness, detail, guidance and traceability to the architectural evaluation. This kind of method is particularly important for a safety-critical system where the safety property needs to be maintained and enhanced in every aspect of the system development. Here, we define an approach especially for safety-critical systems. The approach will use the complete views of the system in order to provide completeness and detail, and follow the hazard analysis methods in order to provide guidance and traceability.

The complete views of a system refers to the “4+1” view model of software architecture developed by Kruchten [23]. The “four views” are the logical view, process view, physical view and development view. Each view describes the different levels of software structure from four different perspectives of system development. The “plus one view” refers to scenarios which represent the behavioral instances of the software’s functionality.

Hofmeister, Nord and Soni [6] took the view model one step further. They identified four different views (conceptual view, module view, code view and execution view) which are used by designers to model the software and kept the “plus one view” (the scenarios) to describe the system behaviors associated with each of the four view. The conceptual view describes the high-level components and connectors along with their relationships. The module view describes the functional decomposition of the software. The code view organizes the source code, libraries, etc. The execution view deals with dynamic interactions between the software and its hardware or system environment. Each view deals with different engineering concerns. The safety-significant requirements will be mapped from the upper-level view into the lower-level view and be further refined in the context of that view. By segregating the system into four views, it gives a systematic way of making the system’s structure visible which helps to understand the domain in the early design stage, to identify the safety-significant requirements, and to make sure nothing important is missing.

In our approach, there are two directions as shown in Figure 13 below.
The vertical direction views the system from four different perspectives (the four views of [6]). Thus, the safety-significant requirements can be carried on and enhanced from global (the high-level view) to local (the detailed development). In this way, we can derive relatively detailed and completed sets of scenarios for each level view and those scenarios can later be input into an architectural evaluation method which will produce the architectural decisions for the development of the system.

Before the steps in the horizontal direction are to be carried out, the approach will perform a global analysis which derives an overall safety checklist. A utility tree which is similar to that of ATAM for quality attributes will be constructed based on the checklist. The checklist populates the first level nodes in the utility tree and guides the derivation of the safety-significant requirements.

After that, two steps will be used in each view to derive the set of scenarios. First, SFMEA is used to work forward to identify potential hazards and prioritize them. The more significantly a hazard will affect the system development, the higher priority the hazard has. Furthermore, the higher priority hazards will be picked up and further explored in the next view. Thus, it offers sound guidance for engineers to focus on the aspects of the system which are safety-critical, and provides traceability from the high level views to lower-level views and vice versa.

Second, as described in Chapter One, SFTA is used to work backward to reason about the safety issues of requirement or design documents statements. The paths in the fault tree lead from the leaf nodes (basic events) to the top hazard. A minimal cut set of a fault tree represents the basic
events that will cause the top hazard and cannot be reduced in number —— that is, removal any of the basic events in the set prevents the occurrence of the top hazard [9]. There are more than one minimal cut sets in a fault tree. Based on each minimal cut set of the fault tree, we can derive scenarios which will be the input to future architectural evaluation by representing each cut set as sequence of system behaviors starting from the basic events to the top hazards. Because the minimal cut sets will cover every possible path in the fault tree which will cause the top hazard to occur, a relatively completed set of scenarios can be derived. Thus a relative complete architectural evaluation input can be achieved.

The rest of the chapter is organized as follows. Section 2 describes the related work. Section 3 provides an overview of the approach. Section 4 explains the detailed steps of the approach and discusses its results in section 5. Section 6 illustrates the approach by applying them to a case study and presents some concluding remarks and describes the use of the results of the approach in the future software architectural evaluation.
2. Related work

The research work reported here is based on the related work of four areas. They are safety analysis, product families, software system architectural views, and software architectural evaluation.

In the areas of safety analysis and product families, most of the related work is reported in Chapter One, section 2. A safety analysis technique which is not mentioned there but is used in Chapter Two is Software Failure Mode and Effect Analysis (SFMEA). SFMEA is an extension of hardware Failure Mode and Effect Analysis (FMEA) which has been standardized [26]. SFMEA is well-documented [7] though there is no standard to guide performing SFMEA. We adopt the SFMEA method here to identify and prioritize hazards for the targeted system.

In the area of software system architectural views, Kruchten [23] defined the “4+1” view model. The “four views” are the logical view, process view, physical view and development view. Each of the four views deals with different perspective of engineering concerns of system development. The “plus one view” refers to the scenarios which describe system behaviors. A related, more recent approach by Hofmeister, Nord and Soni [6] is used here. The later approach introduced four new views. They are the conceptual view, module view, code view and execution view. These later four views are more closely related to the software design model. We base our approach on the later four views for system decomposition and keep the scenario derivation to describe the system behaviors.

In the software architectural evaluation area, Shaw and Garlan [4] defined different architectural styles. When there is more than one architectural style can be chosen, how to select the best one is a question which needs to be asked. The Architecture Tradeoff Analysis Method (ATAM) [5] is a way to evaluate different architectural styles by trading off the advantages and disadvantages of using each of the available styles. The ATAM is a quite high-level method. The input to ATAM is a set of scenarios. Unlike our approach, the scenarios used in ATAM are derived from the brainstorming of experts and thus may lead to incompleteness and lack of detail. In our approach, we instead define a systematic way to derive scenarios and thus provide relative completeness, detail, guidance and traceability.
3. Overview of the approach

There are three key concepts on which the approach is founded. First, the four architectural views of system (conceptual, module, code and execution) provide a design way to view the system from four different perspectives and answer different engineering concerns in each view. Second, the safety-significant requirements define the overall safety factors for the system and provide the guidance for the engineers to further explore those factors. Third, the hazard analysis methods (SFMEA and SFTA) for safety-critical systems elicit and prioritize scenarios in terms of stimuli and responses. The derived scenarios provide a visualization of the safety quality of the system into testable requirements which can serve as the input to the later architectural evaluation. This evaluation affects the key architectural decisions which will in turn impact the safety quality of the system. The prioritized scenarios will lead the engineers to further map and refine them in the context of different views and also guide the architects to make appropriate architectural decisions during the architectural evaluation. Armed with the four perspectives of system views and relatively detailed and completed scenarios, a software architectural evaluation can be carried out based on these inputs.

Figure 14, the overview of the approach

Figure 14 above provides an overview of the approach. The vertical direction describes the decomposition of the system into four views and maps the safety-significant requirements into each
view. This is a process where both the knowledge of the system and the safety requirements are detailed and enriched. It also illustrates how to map the safety requirement into each view and the interaction between views.

There are two steps in the vertical direction. The first step is the global analysis. The global analysis is to identify the overall safety factors which will be further explored and refined into the four system views. This is achieved by applying the accumulated past experience checklists which are derived from various types of former developed projects and the four architectural views' safety category [6]. The global analysis will result in a utility tree with the 'utility ---- safety' as the root node and the overall safety factors as the second level nodes. The remaining lower-levels' nodes will be scenarios derived for each view during the horizontal direction of the approach to refine and reinforce the overall safety factors. Therefore, the scenarios are the visualizations of the overall safety factors. The utility tree provides a top-down mechanism for directly and efficiently translating the safety goal into concrete scenarios. A sample utility tree is shown in Figure 15.

```
Input and output
  Interface
    S1: A display error causes wrong data to be input
    S2: Some characters fail to be displayed in specific machines

State completeness
  Trigger event completeness

S: Scenario
```

**Figure 15. A sample utility tree**

The second step of the vertical direction is the four views' system decomposition. The four views capture the engineering concerns of the system from four different perspectives. The resulting overall safety factors from the global analysis will be mapped into each view and refined into detailed concerns in the context of each view. In each view, new safety concerns may also be identified and refined. Furthermore, these newly identified safety concerns may be mapped into later views when
they have impact on those views. There are feedback and interaction between views. Though most of
the engineering concerns of each view are independent of other views, some may be affected by the
views that are designed later. Thus, the later views may reflect constraints back to the earlier views.
The relationship between views is bi-directional and iterative. Figure 16 below shows the relationship
of the four views.

![Figure 16. The relationship of the four views](image)

The first two views (conceptual and module) include design issues which involve the overall
association of system capability with components, modules and interconnections between them. The
third view (code) includes design issues which involve algorithms and data structures. The fourth
view (execution) contains the design issues which are related to hardware properties and system
environment such as memory maps, data layouts.

In the conceptual view, module view and code view, some design concerns of the former one
will be mapped into the later one and be further refined, while the execution view may have
interconnections with any of the other three views because the software concerns of those three views
will be affected by the hardware environment and also put constraints back to the hardware
environment at the same time.

Refining the safety requirement based on the four views provides a way to define and
describe the system goal (safety) in relative detail. The factors needing to be reinforced in each view
are picked up from the result of global analysis. In addition, in each view, some new factors will be investigated if they will play additional important roles within this view. For example, in the code view, the language choices will be introduced and investigated to satisfy the needs of the system development. The 4-view decomposition provides a way to understand the system systematically and a structured context where the processes in the horizontal direction can be performed.

In the vertical direction, we have mentioned that scenarios will be derived for each view to detail the overall safety requirement. How the scenarios are derived and prioritized is the task which will be fulfilled in the horizontal direction.

There are two steps in the horizontal direction which will be performed for each view. These two steps are the hazard analysis methods (SFMEA and SFTA) for safety-critical system which work forward and backward to identify safety requirements from the original requirement documents. The way we use hazard analysis methods for achieving the safety quality of the system is to identify the potential hazards of the system and then make the system safe by preventing those hazards from happening. Thus, the scenarios derived from hazard analysis methods are actually the misuse cases which may occur in the system in special conditions and thus compromise the safety of the system. We introduce the two steps in detail as follows.

First is the forward hazard analysis ---- the SFMEA. As explained in Chapter Two, SFMEA works forward from the failure modes to possible effects to identify the potential hazards and prioritize them. The overall safety factors resulting from the global analysis will guide the hazard identification here. For each view, the overall safety factors will be mapped into this specific view by checking the items of the factors one by one and identifying the hazards related to each item within this view. The identified hazards will be prioritized according to the potential effects they may cause. The high-priority hazards will be mapped into the next view and further explored and refined. Thus, the traceability can be achieved from early views to later views and vice versa by tracing the columns which record the derived hazards in SFMEA tables for the four views. After the SFMEA is performed, we will have a set of prioritized hazards for each view. Also, for each hazard, the mechanisms provided to prevent or mitigate the hazards are usually additional safety requirements or new constraints on existing requirements. Those hazard prevention mechanisms will feed forward into the later detailed design processes and help identify the architecturally significant requirements for aiding derivation of the later architectural decisions. The entries in SFMEA are depicted in Table 1.
Table 1. The entries of SFMEA

<table>
<thead>
<tr>
<th>Item</th>
<th>Failure mode of failure</th>
<th>Possible effect</th>
<th>Priority</th>
<th>Hazard reduction mechanism</th>
<th>Future Architectural decision</th>
</tr>
</thead>
</table>

The second step is the backward hazard analysis —- the SFTA where the scenarios are going to be derived. As described in Chapter Two, SFTA works backward to reason about the existence of a specific combination of requirements which may lead to the root hazard. In each view, SFTA will take the identified hazards with high priorities in SFMEA as the root nodes and perform fault tree analysis on them. Each minimal cut set from the leaf nodes (the basic events) to the root hazard will be the potential requirement problems which may cause the hazard to occur. After the fault tree has been constructed, every minimal cut set in SFTA will be mapped into the scenarios. The basic events in the minimal cut set will be the stimuli of the scenarios. The paths from basic events to the root node represented by the intermediate nodes in the fault tree will be the series of actions in the scenarios after the stimuli have been triggered. Finally, the root hazard will be the response in the scenarios. Because every minimal cut set will be considered during the scenario derivation, a relatively complete coverage will be achieved. The derived scenarios will become the lower-level nodes in the utility tree. For example, in the utility tree in figure 3, the overall safety checklists such as input and output, interface, etc are the first level nodes. The S1 and S2 are two scenarios recorded as the lower-level nodes under the “interface” factor of the overall safety checklists.

The process of deriving scenarios from the minimal cut set of fault tree is shown by the following example. Portion of a fault tree for a safety-critical system —- the insulin delivery system [25] is shown in Figure 17. The dotted lines in the figure are those fault tree parts which are omitted here for simplicity.
From the path shown in the fault tree above, a scenario for the code view can be derived: an algorithm error happens during a sugar computation, thus causing an incorrect sugar level measured; the system’s response is to administer an incorrect insulin dose to patient.

After we have introduced the vertical and horizontal directions as well as the activities within each direction of the approach, we summarize the entire approach as an algorithm presented below.

1. Perform a global analysis for the entire system and identify the overall safety factors.
2. Within the domain of each view, use SFMEA to derive new hazards by applying the overall safety factors as guideline and to refine high priority hazards input from earlier view.
3. Prioritize hazards based on the potential safety effects they may cause and define the mechanisms which will be used in future design to prevent the hazards from happening.
4. Use SFTA to apply fault tree analysis for each high priority hazard
5. Derive scenarios by mapping each minimal cut set of a fault tree into a scenario.
6. Input the derived scenarios into utility tree as the lower-level nodes.
7. Repeat steps 2 through 6 for each of the conceptual, module, code and execution views until the scenarios have been fully derived for each of the views.

The result will be that a complete utility tree has been constructed with 'safety' as the root node, the overall safety factors as the second-level nodes, and all the scenarios as the lower-levels' nodes.

We will illustrate the above steps of the overall approach in the following two figures. Figure 18 outlines the overall approach. Step 1 of the above process, that is, the global analysis, will be performed before the four-view system decomposition starts. The overall safety checklists will become the first level nodes of the utility tree and also they will be mapped into each of the four views. After all the analysis activities (step 2 through step 6) within each of the four views have been taken, the output scenarios will become the lower-level nodes in the utility tree. Thus, a complete utility tree will be constructed as described.

Figure 19 explains the inside activities of each of the four views. Those activities have been described in detail in the above step 2 through step 6. The overall safety checklists derived by the global analysis will become the input to each of the four views together with the high-priority hazards of the earlier view. (But note that there will not have been any high-priority hazards input to the conceptual view because it is the first view of the four views.)

The overall safety checklists will serve as the guideline for SFMEA to derive potential hazards for each view. At the same time, the high-priority hazards input from earlier view will be further refined. All the derived hazards will be prioritized according to the possible effects they may cause. The high-priority hazards of each view will be further analyzed by applying SFTA. After the fault tree has been built, every minimal cut set of the tree will be mapped into a scenario. Finally, the derived scenarios will be input into utility tree and become the lower-level nodes of the utility tree.
Global analysis to derive the overall safety checklists. Output overall safety checklists to become the first level nodes in utility tree.

The four-view system decomposition. Output scenarios to become the lower level nodes in utility tree.

Utility tree

Figure 18. The outline of the approach

High priority hazards from early view

The overall safety checklists

Apply SFMEA to derive hazards

Prioritize hazards and define hazard prevention mechanism

Apply SFTA to construct fault trees for high priority hazards

Map paths of SFTA into scenarios

Scenarios become the lower level nodes in utility tree

Future architecture evaluation

Figure 19. The activities taken within each view

After the two-dimensional approach has been performed, we will have two products in hand. One is a completed utility tree. In this utility tree, the overall safety checklists have been refined and detailed. The scenarios derived from each view will be grouped under the specific item of the overall safety checklists when the scenarios are related to this item. Thus, from the root node to the leaf nodes in the utility tree, the safety goal of the system has been detailed. It provides a basis for evaluating future software architectural decisions by checking how much and how well they are designed to serve the safety goal.
Another product is the SFMEA tables for the four views. From those tables, a clear mapping from the high-priority hazards of one view to the next can be traced. For example, a hazard of the conceptual view says: the insulin dose was mis-administered. Because this hazard has high priority, it is input into the module view and further refined into two hazards there. One is “the insulin administered is overdose”; another is “the insulin administered is underdose. In this way, the high-priority hazards of the conceptual view can be traced into the module view. Also, the hazard prevention mechanisms which are included in the SFMTA provide good guidance for the architect when architectural decisions need to be made to fulfill the safety quality of the system. On the ground of these two results, an acceptable and feasible software architectural evaluation may be able to be carried out in the future.

The contributions of the technique we have presented here are: (1) to provide a structured and systematic way to decompose safety requirements into detailed and testable scenarios by viewing the system from the four architectural perspectives; (2) to identify the potential architecturally significant requirements in terms of the safety point of view by listing the hazards in SFMEA and defining the prevention mechanisms for the hazards; and (3) to support future software architectural evaluation by constructing an input to it with relative detail, completeness, guidance and traceability.

In the next several sections, we will discuss the process of the approach in detail by applying it to a case study. The case we will use here is the insulin delivery system [25]. The insulin delivery system monitors the blood glucose level of diabetics and automatically injects insulin as needed. We selected this system based on its having a suitable size to be used as a case study and its safety concerns.
4. Global analysis

The Global analysis is to identify the overall safety factors which may have an impact on the system safety. If there are potential hazards involved in those factors, the safety of the system may be compromised. Furthermore, some of these hazards are not likely to be identified, caught and prevented during the testing stage of the system development. To avoid the expensive rework, even the more expensive catastrophic consequences, these factors must be addressed from the beginning of development.

The purposes of the global analysis are to identify those factors which may influence the safety of the system and to guide the further processes of the approach. Global analysis starts before the four-view system decomposition begins. The results of the global analysis will become the input to each of the four views. There are three activities in the global analysis.

(1) **Identify and describe the safety factors.**

The factors needing to be identified and considered are those which may affect the safety of the system and which must be satisfied during development. There are two ways to perform the factors’ identification.

One way is to use the accumulated experience checklist from former projects. The experience checklist usually comes from the brainstorming session of experts and continues to be extended and enriched during project development. There are two advantages of using the experience checklist especially to derive the safety factors when a similar new system is going to be built. First, the valuable old experience can be reused and part of the rework can be saved. Second, those problems which have been identified and avoided during the previous project’s development can be easily picked up and prevented in the development of the new system.

Though the experience checklist may cover most aspects of the system, the new system may bring in new concerns. In order to cover these new concerns, a brainstorming session may be hold for experts to derive new factors. To what extent the new concerns can be derived entirely depends on the results of the brainstorming session. Thus the completeness will be a problem. Particularly, when a totally new system is going to be built, there hardly exists any old experience checklist which can be used as the basis to derive the safety factors.

Another way is to import the four architectural views’ safety category defined by [6]. The four architectural views’ safety category is those safety concerns identified in each of the four
architectural perspectives. Groups of the factors in the category are the human-computer interface, input and output, etc. Directly introducing the safety category to derive the overall safety factors during global analysis provides two benefits. First, the safety category itself is clearly described and organized, thus it presents a starting point for a safety checklist to which additions may be made as we discover the new concerns involved in the system. Second, the safety category represents the general safety concerns and thus can be easily applied to different kinds of systems. Third, the safety category includes not only the software aspect of the safety concerns but also those hardware or management concerns which may have safety impact on the system as well. For example, a management concern says: Policies need to be defined in order to regulate the operators’ behaviors. A violation of those policies may cause hazards in this system.

From the above advantages of using either the accumulated experience checklist or the four architectural view’s safety category during the global analysis, we define a process which combines both of the above ways to identify and describe the factors.

In our process, the four architectural views’ safety category [6] will be taken as the primary way to identify the overall safety factors in the global analysis. The accumulated experience checklist will work as the complement to identify the management factors which deal with staff skills, development schedule, etc and the hardware constraints which include what hardware platform will be used, the equipment quality, etc. Thus, the derived overall safety factors will include all the three aspects (software, hardware, management) of the system development. Therefore, the results of the global analysis achieve a relatively complete coverage which may affect the accuracy of the further analysis processes in the approach.

(2) Identify the part of the safety factors to satisfy the needs of a specific system

As we have mentioned above, the overall safety factors will be identified and described by using the four architectural views’ safety category as the basis and the accumulated experience checklist as the supplement. The resulting set of overall safety factors are relatively completed and general so as to be applied to various kinds of systems. However, some of the factors may not be applicable to a specific system. For example, a wind speed monitoring system will not take any input from keyboard or any other ways manipulated by a human operator. Thus, the “human-computer interface” safety factor will not be applicable in this case. Thus, it is not efficient to consider all those safety factors while some of them may not relate to the system.
Therefore, we will extract the part of the identified overall safety factors to satisfy the needs of the specific system which is going to be built. Only those factors which are related to the system will be kept. By this way, we scale down the quantity of the overall safety factors and only focus on those factors which are applicable to the system.

(3) **Analyze the impacts of the safety factors**

After we select those safety factors which relate to the system, we will analyze the potential impacts of the safety factors on the system development. The analysis will be performed by asking the question “if the factor involves problems, how severely will it affect other factors and hence the system development?” Every safety factor will be categorized as high, medium and minor according to the effect it may have on the system. For example, if there are problems in the “input and output” factor such as an output generates wrong value, it may affect other pieces of the software in the system which take the wrong output as input and thus perform incorrect operations. Therefore, the “input and output” factor will be assigned a label of “high”. While if there is a possible power shut down in the “power supply” and there is a backup battery, the “power supply” will be categorized as “minor”.

By analyzing the impact level of the safety factors, we can order them from high priority to low priority. The higher the impact level is; the higher priority the factor will have. Thus, during the system development, more attention can be paid and more resources can be allocated to those factors which have high priorities. It provides an efficient way for engineers to focus on the most important parts of the system during development.

(4) **Construct the utility tree**

After the set of overall safety factors with different priorities are derived, they will become the first level nodes below the root node in the utility tree and be ordered from high priority ones to low priority ones. A sample utility tree is shown in Figure 20. The rest of the utility tree will gradually be constructed as the remaining steps of the approach. The safety factors will also be input into the four-view system decomposition and further refined into scenarios.

When the global analysis is complete, we will have characterized the safety factors, analyzed the impacts of them on the system development, prioritized the safety factors for ensuring completeness and guidance of the further processes in the approach and constructed the utility tree with the overall safety factors as the first level nodes.
We illustrate the activities of the global analysis below by applying them to the insulin delivery system [25]. The insulin delivery system uses sensors to monitor the patient’s blood sugar level, and then calculates the amount of the insulin needs to be delivered. The controller will send the command to the pump to pump the set amount of the insulin. Finally, the insulin will be sent to the needle and inject into patient’s body. Thus, the insulin delivery system is an embedded system and uses several equipments to fulfill the task.

We will follow the four activities in the global analysis described above step by step during performing the analysis process on the insulin delivery system.

Step one will identify and describe the safety factors. We import the four architectural views’ safety category defined by [6] and use the checklist of an advanced imaging solution system ---- IS2000 [6] as the complement. A set of safety factors is derived as follows.

Software:
- Human-computer interface
- Input and output
- Input and output Trigger
- Output specification
- Output to trigger event relationships
- Specification of transitions between states
- Performance

Management:
- Product cost
- Staffs skills
- Development schedule
- Development budget

Hardware:
- General-purpose hardware (processor, network, memory, disk)
- Equipments quality
Step two will take the set of the safety factors derived in step one as input and pick up those that relate to the insulin delivery system. Based on the needs of the insulin delivery system, the following safety factors will be selected as relevant to this system.

Software:
- Human-computer interface
- Input and output variable
- Trigger event
- Output to trigger event relationships
- Specification of transitions between states

Management:
- Staffs skills
- Development schedule
- Development budget

Hardware:
- Equipments quality

Step three will analyze the possible impacts of the above safety factors on the system. Those software factors will interact with each other. If any of them has problems, the problems may propagate among other factors and affect the safety of the system. Thus, we assign the impacts of the software factors as high. Among the management factors, the staffs skills may have effect on the system development quality and thus be assigned as medium. How important the development schedule and budget are depend on the company’s condition. In this case, we assume that the schedule and budget will not be problems and assign them as minor. Because the insulin delivery system will be put in the patient’s body to operate, the quality of the equipments such as sensors, needle will be critical. Thus, we assign the impact of the hardware factor as high. A table can be constructed to display the set of the safety factors and levels of their potential impacts on the system. The table is shown in Table 2.
Table 2. The table of the overall safety factors

<table>
<thead>
<tr>
<th>Safety factor</th>
<th>Impact level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human-computer interface</td>
<td>High</td>
</tr>
<tr>
<td>Input and output variable</td>
<td>High</td>
</tr>
<tr>
<td>Trigger event</td>
<td>High</td>
</tr>
<tr>
<td>Output to trigger event relationships</td>
<td>High</td>
</tr>
<tr>
<td>Specification of transitions between states</td>
<td>High</td>
</tr>
<tr>
<td>Staffs skills</td>
<td>Medium</td>
</tr>
<tr>
<td>Development schedule</td>
<td>Minor</td>
</tr>
<tr>
<td>Development budget</td>
<td>Minor</td>
</tr>
<tr>
<td>Equipments quality</td>
<td>High</td>
</tr>
</tbody>
</table>

The above table is a guideline for engineers during the later steps of the approach. Based on the impact level of an individual factor, engineers can decide to what extent they need to explore this factor. Thus the development resources can be efficiently allocated to the most significant factors.

Step four will construct the utility tree from the above safety factors’ table. All the safety factors will become the first level nodes in the utility tree and they will be ordered from top to the bottom according to their impact levels. High level impact factors will be placed first, and then the medium and the minor level ones. The utility tree is depicted in Figure 20.

After we perform the four activities of the global analysis on the insulin delivery system [25], the overall safety factors have been derived, prioritized and clearly described. Also, the first level of
the utility tree for the insulin delivery system has been built. These two results provide a basis for the further processes in the approach to be carried out. We will continue to illustrate the approach on the insulin delivery system in the following sections.
5. The four-view system decomposition

The four system views [6] deals with system development concerns from different perspectives. Most of these concerns are independent of each view from the other views. However, there are interactions among views. Some concerns of the high-level views may be mapped into low-level views and be further explored there. Also, during handling of low-level views’ concerns, they may put constraints back to the high-level views. The relationships of views have been depicted in Figure 16 in section 2.

5.1 The definitions of the four views

The conceptual view is the highest system level view and thus the first level of the system abstraction. The conceptual view is tied very closely to the application and describes the overall functionalities of the system, so it is less constrained by the hardware and software platforms. The conceptual view treats the functionalities of the system as black boxes and as being relatively independent from detailed software and hardware techniques. The functionalities are mapped into the conceptual view by being represented as *components*. The communications and data exchanges between those functionalities are handled by *connectors*. For example, if the pipe and filter architectural style is used in developing a system, the pipes are the components and the filters are the connectors in the conceptual view.

The module view takes a closer look into the system’s implementation. It deals with how the functionalities of the system can be realized with software techniques. The system functionalities which have been outlined in the conceptual view are mapped into sub-components and sub-systems in the module view. Thus the abstracted functionalities will be decomposed into several functional parts and each provides a specific service to the system. Also, the data exchanges between sub-systems or sub-components will be described in the module view. For example, a balance inquiry functionality of a banking system will be decomposed into several sub-systems in the module view. Figure 21 below shows the decomposed balance inquiry functionality and the message passing between its sub-systems.
Figure 21. The module view of the balance inquiry functionality

The execution view describes the system development in terms of its runtime platforms elements which include software and hardware properties, such as memory usage, processes, threads, etc. It captures how these platform elements are assigned and how the resources are allocated to fulfill the system functionalities. The sub-systems and sub-components of the module view are mapped into the execution view by allocating the software or hardware resources for them. For example, in a client-server system, the client and server will be mapped into two separate processes.

The code view describes how the system is organized in terms of software implementation. In this view, all the functionalities of the system are fulfilled and all the runtime entities assigned by the execution view are instantiated by using a specific programming language. Thus the dependencies among system functionalities, library functions, and configuration files are made explicit in this view.

The purposes of performing the four-view system decomposition are:

1. Breaking down the complicated system structure by separating the different engineering concerns into four views according to the properties of each view defined above. Thus a relatively clear approach can be pursued.

2. Analyzing the engineering concerns independently within each view and identifying their possible effects on other views. Thus, a local problem which may have global effect can be captured.

3. Providing an organized way to map the key issues (the overall safety factors derived from the global analysis) from view to view. Thus, those important issues can be carried out and explored during the entire system development process.
The order used to perform the four views is driven by the engineering concerns of each view. The conceptual view will be performed first because it is the highest level system view. It will be followed by the module view in which the functionalities outlined in the conceptual view will be further refined. After the module view, the execution view and the code view can be carried out iteratively. First, the execution view allocates the software and hardware resources to the sub-systems or sub-components of the module view. Then the code view describes the implementation of those sub-systems or sub-components by using the software and hardware resources assigned in the execution view. While during the system implementation, some specific concerns in the code view may put constraints back to the execution view. For example, a kind of programming language picked up in the execution view may not be able to handle the interface requirement in the code view, and thus may have to be changed to another one.

Within each view, a set of activities will be carried out. These activities are used to instantiate the overall safety factors derived in the global analysis according to the context of a specific view, to perform hazard analysis and prioritizing, and to derive scenarios which may become the input to the later software architectural evaluation. The details of the activities involved in each view will be explained in the following section.

5.2 The activities of each view

As we have mentioned in the overview of the approach, the four-view system decomposition provides a structural way to analyze the system in the vertical direction while the horizontal direction, the activities involved in each view, actually performs the detailed steps of the approach. There are two main activities that need to be fulfilled within each of the four views.

5.2.1 Hazards derivation and prioritizing ---- SFMEA

The overall safety factors derived in the global analysis prior to the four-view system decomposition are those key issues which must be satisfied in the system development. Thus, if there are hazards involved in the overall safety factors, those hazards may cause catastrophic consequences. By partitioning the system development activities and engineering concerns into four different views, it is possible for us to achieve a relatively complete coverage of hazards derivation. After the hazards
have been derived and prioritized according to the potential effects they may cause, the possible hazards prevention mechanisms can be defined. The defined hazards prevention mechanisms may benefit the later detailed design and the prioritized hazards can guide engineers to the further hazard exploration and refinement. The SFMEA format is used here to derive, prioritize and record the hazards. A sample SFMEA table is showed in Table 1. There are two main steps in this activity by following the SFMEA method.

(1) **Identify the hazards and their causes.**

There are two ways of identifying the hazards for each of the four views.

First, instantiate and refine the overall safety factors derived in the global analysis. The overall safety factors will be input into each view and work as the guideline to derive hazards according to the properties of the engineering concerns of that view. The items of the overall safety factors will be applied one by one to the specific view. Those possible hazards which relate to a specific view will be identified. For example, when applying the “input and output” of the overall safety factors to the conceptual view of the insulin delivery system, the hazard “insulin dose is mis-administered” will be identified.

Second, further explore and refine the high priority hazards of the earlier view. Within a view, if a hazard has high priority that means this hazard may cause catastrophic consequence. Thus more attention and more resources need to be put on this hazard to prevent it from happening in the entire life time of the system. The later view will take the high priority hazards in the earlier view as input and further analyze and refine those hazards. By this way, the hazards derivation can be traced from the earlier views to the later views and vice versa. However, note that this way of hazards derivation is not applicable to the conceptual view. Because the conceptual view is the earliest view, thus no high priority hazards of earlier view exist to be used as the input. Take the above hazard “insulin does is mis-administered” of the conceptual view as an example. Because if the insulin delivered is a mis-dose, it may cause patients to die, thus this hazard is classified as high priority and input into the module view. In the module view, this hazard is refined into two hazards according to the context of the view. One is “insulin overdose” and another is “insulin underdose”. If any of these two hazards is identified as high priority, it will be input into the execution view and the code view and be further refined there.

After the hazards have been identified, the possible reasons which may cause the hazards are assessed. The causes of the hazards provide valuable information for engineers to uncover the parts of
the system which may involve problems. By isolating the hazardous parts of the system, engineers can focus on these parts of the system for further hazard analysis and exploration. Thus, the identified possible causes of hazards can serve as guideline for both the hazard derivation in later views and for appropriately allocating the development resources.

(2) Prioritize the hazards and identify hazards prevention mechanism.

After performing the above two ways of hazards identification, a set of hazards will be derived. For each of the hazards, the potential effect it may cause will be assessed. Based on the effect of each hazard, a level of priority will be assigned. The severer the effect a hazard can have, the higher the priority it will have. Here, we use three key words (high, medium and minor) to classify the priority levels. High-priority hazards may cause severe consequence to the system and thus deserve more attentions and more system resources to be spent to prevent them from occurring. According to the engineering concerns of each view, the potential hazard preventing mechanisms will be defined for those high or medium level hazards. Those minor hazards which will not have much safety impact on the system may be left unchanged first and be prevented when there are enough development resources available.

The results of the above two steps will be organized and recorded into a Software Failure Mode and Effect Analysis (SFMEA) table. The identified hazards, the causes of the hazards, the priority levels of hazards, and the possible hazard prevention mechanisms will each become an entry in the SFMEA table. We use the “insulin overdose” hazard in the module view of the insulin delivery system as an example. After performing the above analysis steps, the results are shown in the following SFMEA table as Table 3.

There are four advantages to introducing the SFMEA method here:

First, SFMEA is a widely used hazards analysis method. By following the SFMEA approach, an organized way of hazard analysis can be gained and it is easy to export the results of the SFMEA to aid other analysis methods if necessary.
Table 3. The SFMEA table for the "insulin overdose" hazard in the module view

<table>
<thead>
<tr>
<th>Item</th>
<th>Failure mode</th>
<th>Cause of failure</th>
<th>Possible effect</th>
<th>Level</th>
<th>Hazard prevention</th>
</tr>
</thead>
</table>
| Insulin dose administered | Overdose     | 1. Incorrect blood sugar measured  
2. Blood sugar analyze wrong  
3. Insulin amount computed incorrectly  
4. Insulin delivery controller malfunction  
5. Incorrect amount of insulin pumped. | Injury or death | High  | Rigid quality control of equipments such as sensor, pump, controller to eliminate defectives. Inspection and comprehensive tests are to be used to ensure the correctness of analysis and computation methods. |

Second, each of the four views will have a SFMEA table for itself which records all the hazards derived within this view. Thus, the traceability can be achieved by tracing the evolving of hazards from the high-level views to the low-level views. Also, the origins of hazards can be found by tracing them from the low-level views back to the high-level views.

Third, the priority levels of the hazards showed in the SFMEA table benefit the further analysis activities such as Software Fault Tree Analysis (SFTA) which will pick up the high priority hazards and perform the fault tree analysis until the basic events that may cause the occurrences of the hazards have been found.

Fourth, the last column in the SFMEA table records the possible hazards prevention mechanisms. It provides the possibility of expanding the original SFMEA table to include software architectural decisions which are made to realize those hazards prevention mechanisms in the system development.

5.2.2 Hazard analysis and scenario derivation ---- SFTA

The results of the first activity described above are the derived and prioritized hazards. Those high-priority hazards may contribute to the occurrences of catastrophic consequences, and thus need to be further analyzed in the second activity by using software fault tree analysis (SFTA). As introduced in Chapter One, SFTA works backward from the root hazard to find conditions and events
that will potentially cause the hazard to happen. After the fault tree for a specific hazard has been constructed, the scenarios can be derived and input into the utility tree. There are two steps in this activity.

5.2.2.1 Hazard analysis and fault tree construction

The priority levels of the hazards recorded in the SFMEA table guides the fault tree analysis. Each high-priority hazard of a specific view becomes the root hazard of a software fault tree analysis. Whether to expand the fault tree analysis to include medium or even minor-priority hazards depends on the system needs and the availability of development resources. The software fault tree analysis will be carried out within the context of a specific view. Therefore, for each view, its SFTA analysis will only trace down to the requirement problems which are at the same level of detail as that of this view. However, the resulting fault trees may be subject to further exploration in the next view if necessary. For example, one leaf node of the fault tree of an earlier view may become the hazard in a later view and thus be further refined there. For example, consider the high-priority hazard “insulin overdose” of the insulin delivery system [25] in the module view shown in Table 3 as the root hazard. After performing the fault tree analysis, the resulting fault tree is as shown in Figure 22. The dotted line represents the omitted part of fault tree.

From the above fault tree, we can see that the analysis will not exceed the boundary of the module view. However, some leaf nodes may directly become the hazards in the code view. For example, the leaf node “sugar computation error” and “insulin computation incorrect” will be the two hazards in the code view and further analyzed there.

By performing the SFTA in this way, we can consider the possible hazards and their causes within a specific view. This allows the potential problems existing within each view to be isolated and treated independently. Also, the fault trees are adaptive and expandable in that the nodes of the fault trees in the earlier views can be further analyzed in the later views. Thus, the involvement of the same hazards in different views can be traced.
5.2.2.2 Derivation of misuse scenarios and utility tree construction

After the fault trees for high-priority hazards have been constructed in the first step, the misuse scenarios [26] are ready to be derived and the utility tree to continue to be built. In this step, we will perform two tasks.

The first task is to derive the misuse scenarios from the software fault trees. The misuse scenarios are those system behaviors which should not occur and which may cause the root hazards. The method used to derive those misuse scenarios is based on the minimal cut sets of the fault trees. The minimal cut sets [Leveson] represent the basic events (the leaf nodes) that will cause the top hazard and can not be reduced in number. For example, in the fault tree shown in Figure 22, every leaf node will be contained in a minimal cut set because there are only OR gates in the fault tree. However, if the fault tree involves AND gates or other special gates, then a minimal cut set may include two or more leaf nodes.

The order of the occurrences of the events represented in the fault tree, that is, the lower-level nodes should appear earlier than the higher-level nodes, is ready to describe the behaviors of the system in terms of the scenarios. The paths from the root hazard to each of the minimal cut sets will
be mapped into misuse scenarios. The leaf nodes which are the causes of the hazard will become the stimuli of the scenarios. The behavioral reactions of the system to the stimuli are represented by the sequences of the intermediate nodes along the paths in the fault tree. Eventually, the system's response may be the occurrence of the top hazard. For example, one of the minimal cut sets is the "sugar computation error" in the fault tree in Figure 22 as example. A misuse scenario can be derived based on the path from this leaf node to the root hazard. The derived misuse scenario is: A sugar computation error causes incorrect sugar level measured; the system response is to administer an incorrect amount of insulin dose.

By deriving the misuse scenarios based on the above method, we can achieve the following advantages:

1. Relatively complete coverage. Every path in the fault tree will be considered and mapped into scenarios when applicable. Thus, a more complete and structural scenarios derivation mechanism can be achieved rather than just using brainstorming sessions.

2. Better understand the system behaviors. The order of events preserved in the fault trees describes the steps of the system behaviors. Thus, the derived scenarios represent the sequences of system actions from stimuli to the possible hazardous response.

The second task is to continue to construct the utility tree. We have introduced earlier how the utility tree starts to be built after the overall safety factors have been derived in the global analysis. Each of the overall safety factors is realized into each of the four views by hazards and scenarios derivation. The hazards derivation is performed as the first activity in the analysis process of each view. During performance of the second activity, the high-priority hazards will be picked up and instantiated into misuse scenarios by using SFTA.

In each view, after the misuse scenarios have been identified, they become the lower-level nodes in the utility tree. Basically, the misuse scenarios of each view are grouped under the specific factor to which they relate at the same level of the utility tree. However, a hierarchical relationship may exist between some misuse scenarios belonging to different views. For example, the stimulus of a misuse scenario in the earlier view may become the system response of a misuse scenario in the later view. For example, the stimulus of the misuse scenario (A sugar computation error causes incorrect sugar level measured; the system response is to administer an incorrect amount of insulin dose.) in the module view is "sugar computation error". It becomes the hazard in the code view and a fault tree will be constructed for this hazard. During the misuse scenarios derivation for the hazard
"sugar computation error" in the code view, all those misuse scenarios derived from the fault tree with this hazard as the root node will be grouped under the earlier misuse scenario in the module view. Figure 23 shows the continually building utility tree in each view and the possible hierarchical relationship between misuse scenarios.

Figure 23. Part of the continually building utility tree

The utility tree which will be continually built in each view by following the above process represents the results of the analysis activities in terms of misuse scenarios together with their relationships with the overall safety factors. Furthermore, the utility tree provides a graphic overview of how the overall safety factors are realized and instantiated into detailed system behaviors (the misuse scenarios) in each view. Also, the utility tree is able to describe the possible hierarchical relationships between misuse scenarios of different views.

From the analysis steps in the second activity introduced above, three benefits can be gained by using SFTA here:

First, a complete forward and backward hazard analysis method can be achieved by combining the SFTA in the second activity and the SFMEA in the first activity. This method provides completeness by considering every possible cause of the hazards during software fault tree analysis. Also, the flexibility can be gained by deciding how detail the SFTA will go based on the available development resources and the system’s needs.
Second, SFTA provides a clear picture of how the series of system behaviors may cause the hazards to occur. Thus, the possible hazards prevention mechanisms may be derived by figuring out how to break the paths which lead to the hazards.

Third, the nodes of the fault trees in earlier views may be subject to further exploration in the later views. Thus, the fault trees of the earlier views are to be expanded in the later views. In this way, the traceability of the hazard involving from the earlier views to the later views can be achieved.
6. Discussion of results

In section 3 and section 4, we have detailed the steps of the approach in both the vertical and the horizontal directions. In the vertical direction, first, the global analysis outlines those overall safety factors that need to be satisfied throughout the entire life time of the system. Then the four-view's system decomposition provides a structured way to better understand different aspects of the engineering concerns during the system development. In the horizontal direction, the detailed steps of analysis actually take place in each of the four views. Those analysis steps realize and instantiate the overall safety factors into the context of each view. After that, detailed mechanisms which are used to fulfill those instantiated overall safety factors can be identified. The evaluation of how well a specific software architecture is in terms of satisfying those instantiated overall safety factors may be carried out.

After performing the entire process, we will have two main products.

(1) The SFMEA tables.

The analysis produces a SFMEA table for each of the four views. The resulting four SFMEA tables provide traceability to the origins of the identified the hazards. By checking the hazards recorded in each table, we can trace the hazards involved from the earlier views to the later views, and where the hazards originate from the later views to the earlier views.

The four SFMEA tables support reuse. The possible hazard-prevention mechanisms that are identified serve as reusable assets during the detailed system design stage. There the engineers need to embed the safety properties into the system development to preventing the hazards from happening.

The four SFMEA tables are expandable. A new column can be added into the SFMEA table to include software architectural decisions which are made to prevent the hazards and thus satisfy the safety of the system. Those architectural decisions are subject to be evaluated and selected in the future software architectural evaluation process.

(2) The Utility tree.

The utility tree starts to be built in the global analysis stage and continues to be constructed in each of the four views. After the construction of utility tree has been finished, the completed utility tree reveals a top-down structure of how the first level nodes – the overall safety factors are detailed
and instantiated into misuse scenarios of each view. The utility tree also shows the hierarchical relationships among those scenarios.

The utility tree provides a visible way to illustrate how the overall safety factors may be violated by listing those misuse scenarios that can occur during the system development and that may cause catastrophic consequences. By comparing how many misuse scenarios are listed under each of the overall safety factors, those factors involving more problems can be easily highlighted. Thus, the guidance can be provided for engineers to pay more attention to those highlighted factors during system development.

The utility tree serves as an input to future software architectural evaluation. As mentioned above, the SFMEA tables can be expanded to include architectural decisions. How to rank one architectural decision over another is the key question needing to be answered during architectural evaluation. The misuse scenarios in the utility tree serve as the input to this architectural evaluation process. By evaluating how many misuse scenarios it can prevent or how efficient it is in preventing misuse scenarios, a better architectural decision can be selected.

Besides the above two main products which can be gained from the approach, there is a valuable byproduct — the fault trees constructed in each view from SFTA. During the analysis, those fault trees work as a bridge to link the hazards in the SFMEA to the misuse scenarios in the utility tree. Later, those fault trees can serve as a reusable asset for product family software. When another family member needs to be analyzed, the fault trees for those same hazards can be reused. Thus some development resources can be saved.
7. Sample Application: the insulin delivery system [25]

In the earlier sections, we have introduced the detailed steps of the approach and the resulting products after the approach. In this section, we will further illustrate the approach by applying it to a case —— the insulin delivery system [25].

For the sake of simplicity, we will apply each step of the approach in some selected views instead of each view. However, the results are still understandable by revealing the connections of steps and the consistency of the approach.

7.1 Perform the global analysis

In section 3, the global analysis, we have listed those general safety factors collected from former experience checklists of various types of systems. In performing the global analysis for the insulin delivery system, the unique characteristics of the system will be studied. The insulin delivery system is an embedded system combining both the hardware equipment and the software systems. The hardware equipment such as the sensors and the needle are the physical means for the blood sugar level to be sensed and for the set amount of insulin to be pumped and injected into the patient's body. The software systems serve to support the functionalities of the hardware equipment and ensure the proper operation of them. Based on the properties of the insulin delivery system, a set of overall safety factors which are related to the system has been picked up and shown below. We have seen the same example in the section 3.

Software:
- Human-computer interface
- Input and output variable
- Trigger event
- Output to trigger event relationships
- Specification of transitions between states

Management:
- Staff skills
- Development schedule
- Development budget
Hardware:
- Equipment quality

After the overall safety factors are identified, the utility tree will start to be built. The overall safety factors become the first-level nodes in the utility tree as was shown in Figure 20.

7.2 Perform the four-view’s system decomposition

From the global analysis, the overall safety factors are derived. Next, the system will be decomposed based on the four different perspectives of engineering concerns. Within each of the four views, the analysis activities (SFMEA, SFTA and misuse scenarios derivation) which have been introduced in section 4 will be carried out. Also, the utility tree will continue to be built.

In this section, we will apply the four-view’s system decomposition to the insulin delivery system. However, we will not describe every activity in every view. By this way, the idea of the approach can be clearly illustrated and easily understood without exposing too many details while the coherence of steps of the approach will still be kept.

7.2.1 The conceptual view of the insulin delivery system

The conceptual view is the first-level abstraction of the system. It reveals the overall structure and the primary functionalities provided by the system. Though it is quite high level and less constrained by the software or hardware platforms, it can still have impact on the system properties such as safety as well. By mapping the factors identified in the global analysis into the conceptual view, we can pay attention to those safety-critical parts of the system in the very early system development stage.

7.2.1.1 Describe the overall conceptual structure

The conceptual view of system decomposition of the insulin delivery system is shown in Figure 24. From the figure, we can find that the system is decomposed into components (which may
be further decomposable) and connectors which connect components. The interconnections between components and connectors are also displayed. The components will be regarded as black boxes which function independently. All the connections and message passing between components will be controlled by the connectors and the interfaces between a component and its connector.

![Diagram of the insulin delivery system](image)

**Figure 24** The conceptual view of the insulin delivery system [25]

The above figure depicts the conceptual view of the insulin delivery system in terms of components and connectors. Here, the connectors are abstracted as arrows pointing from one component to another. The functionalities of each component is explained as follows:

1. **Needle assembly**: a component used to receive a set amount of insulin from pump and deliver it into patient’s body.

2. **Sensor**: a component used to measure the glucose level in the user’s blood and send the results back to the controller.
3. **Pump**: a component that receives the input describing how much amount of insulin dose from the controller, pumps the according amount of insulin from the reservoir and sends it to the needle reassembly.

4. **Controller**: a component that controls the entire system.

5. **Alarm**: a component used to sound an alarm when there is any problem

6. **Displays**: two display components, one of which shows the latest measured blood sugar, another of which shows the status of the system

7. **Clock**: a component which provides the controller correct current time.

Given the overall conceptual structure of the insulin delivery system, it is easy for engineers to understand the components of the system and their connections. Also, the conceptual structure provides a fist-level decomposition of the system and thus serves as a basis for other lower-level views to be performed.

### 7.2.1.2 Identify the hazards by using SFMEA

After the conceptual view of the system has been defined, the overall safety factors are to map into this view by using SFMEA to derive the potential hazards. The way to identify the hazards is to check the items of the overall safety factors one by one and evoke brainstorming among the participating engineers. Because the conceptual view will include concerns from different stakeholders of the system, such as domain engineers, software engineers, hardware engineers and management people, all the different kinds of stakeholders should participate in the hazards identification processes.

The hazards in the conceptual level view of the insulin delivery system [25] have been identified as follows by applying the SFMEA analysis. We will only list the hazards and those components which are involved. However, we omit the other parts of SFMEA such as the causes of the hazards and the hazard preventions. Here we will not further analyze the hazards and perform SFTA, but just explain how the identified hazards can be mapped into the next view (the module view) and derive scenarios after performing SFTA in the module view. The reason for doing so is to provide simplicity by not delving into very much detail at the beginning of introducing the approach, but maintaining the connection between steps of the approach at the same time.
1. Insulin dose mis-administered: an overdose or underdose amount of insulin is administered.

   Factors concerned: Human-computer interface
   Input and output

   Components involved: Sensor, controller, pump, needle assembly

   Hazard level: Catastrophic

   Priority: High

2. Power failure: one or more components stop functioning due to a failure in power supply

   Factors concerned: Power supply problem

   Hazard level: Catastrophic

   Components involved: all components

   Priority: High

3. Machine interferes with other machines: Machine interferes electronically with other equipments in the environment

   Factor concerned: Electrical interference

   Hazard level: high

   Components involved: sensor, controller, alarm

   Priority: medium

4. Sensor problem: sensor may break down or senses wrong blood sugar level

   Factors concerned: Input and output

   Trigger events completeness

   Sensor and actuator quality

   Hazard level: Catastrophic

   Components involved: sensor

   Priority: High

5. Allergic reaction: Patients may be allergic to the medicine or insulin used by the delivery system

   Factor concerned: Clinic problems

   Hazard level: High

   Priority: Medium
From the above five identified hazards, 1 and 4 have software concerns, 2 and 3 are hardware problems, and 5 is related to the organization policy which decides what kind of medicine and insulin will be used in the system. Since our focus is on the software aspect of the system, we will only further map the two software hazards into the next views.

7.2.2 The module view of the insulin delivery system

The module view takes a closer look at the system’s implementation in software. The conceptual view outlines the overall functionalities of the system in terms of components and connectors but does not describe how they communicate. The module view will decompose the components into sub-components or sub-systems, and thus, the abstracted functionalities will also be decomposed into several functional parts which may each provide a specific service to the system. The goal of module view is to make explicit how those functionalities are mapped to the implementation. Thus, the message passing between one sub-system or functional part and another will be revealed in the module view.

7.2.2.1 Reveal the module structure

To illustrate the module view, we decompose three components (sensor, pump, and controller) of the conceptual view and display the message passing among them. The result of the module view for these three components is shown in Figure 25.

From the figure below, the controller component has been decomposed into three functional parts each providing a service to implement the insulin delivery function. The communications among those functional parts are also displayed. By decomposing the component, revealing the internal functionalities included within the components, and depicting the message passing among functional parts or components, a more explicit view of the system structure in terms of implementation has been provided.
On the basis of the module view of the system, first, the identified hazards with high priorities in the conceptual view will be further refined and explored in the module view. Second, the overall safety factors will be reflected in this view by deriving additional hazards within the context of the module view. Both of these two steps can be fulfilled in the SFMEA process.

### 7.2.2.2 Apply SFMEA to derive, refine and prioritize hazards

Given the module view of the system, SFMEA will be used to work forward to derive the new hazards or to refine the high priority hazards from the conceptual view and prioritize them. The SFMEA will not only include the new identified hazards but also refine the existing hazards from the upper view. Thus a traceability of hazards analysis has been provided in the SFMEA table when a hazard needs to be traced back to its origin. Hence, the safety quality of the system can be enforced when the hazards are targeted to be prevented based on the unique engineering concerns of the according view. Finally, the trial hazard prevention mechanisms which will be defined in SFMEA may impact the later detailed design of the system such as making the software architectural decisions or evaluating some existing architectural decisions.

The SFMEA tables can be used as a reusable asset in the system’s development process. The identified hazards together with their prevention mechanisms can be reused when a similar system is going to be built which involve the same hazards.

Because our focus is on the software part, we will only refine the software hazards of the conceptual view into the module view. Also, new hazards can be derived by checking the overall
safety factors here. We will apply SFMEA analysis to the portion of the module view we showed in Figure 25. The results of the analysis are in Table 4.

**Table 4. The resulting module view SFMEA of the insulin delivery system**

<table>
<thead>
<tr>
<th>Item</th>
<th>Failure mode</th>
<th>Cause of failure</th>
<th>Possible effect</th>
<th>Level</th>
<th>Hazard prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insulin dose administered</td>
<td>Overdose</td>
<td>1. Incorrect blood sugar measured 2. Blood sugar analyze wrong 3. Insulin amount compute incorrect 4. Insulin delivery controller malfunction 5. Incorrect amount of insulin pumped.</td>
<td>Cause patient to die</td>
<td>High</td>
<td>Rigid quality control of equipments such as sensor, pump, controller to eliminate defectives. Inspection and comprehensive tests are to be used to ensure the correctness of analysis and computation methods.</td>
</tr>
<tr>
<td></td>
<td>Underdose</td>
<td>Same</td>
<td>Same</td>
<td>High</td>
<td>Same</td>
</tr>
<tr>
<td>Sensor</td>
<td>Incorrect data</td>
<td>1. Sensor break down 2. Incorrect data sensed</td>
<td>Cause wrong blood parameters to be output</td>
<td>High</td>
<td>Rigid quality control of sensors. A range check provided to sound an alarm when the data output by the sensor is over-range or under-range.</td>
</tr>
<tr>
<td>Blood sugar analysis equipment</td>
<td>Incorrect analysis results</td>
<td>1. The input blood parameters are incorrect 2. The method used to analyze blood sugar is incorrect</td>
<td>Cause wrong blood sugar analysis results</td>
<td>High</td>
<td>Recheck the input blood parameters by sensor, discard the data if it's not within the normal range. Comprehensively test the analysis method.</td>
</tr>
<tr>
<td>Insulin amount computation</td>
<td>Incorrect amount computed</td>
<td>1. The input blood sugar level is incorrect 2. The computation algorithm is incorrect</td>
<td>Cause incorrect insulin amount computed</td>
<td>High</td>
<td>Set a normal range for input sugar level, sounds alarm when it's out of range.</td>
</tr>
</tbody>
</table>
73

| Insulin delivery controller | Incorrect pump commands sent | 1. The input insulin amount is incorrect  
2. The delivery controller breaks down | Cause incorrect pump control commands sent | High | Comprehensively test the algorithm  
Compare the input insulin with the past history record, sound an alarm when there is any exception. Rigid quality check for the equipment |
| Insulin pump | Incorrect amount of insulin pumped | 1. The input pump control commands are incorrect  
2. Insulin pump breaks down | Cause incorrect amount insulin pumped | High | Comprehensively test the algorithm  
Compare the input commands with the past history record, sound an alarm when there is any exception. Rigid quality check for the equipment |

From the above SFMEA table, we can find that the former two hazards (The insulin mis-administered and sensor problem) in the conceptual view have been refined and four additional hazards have been identified. There are several things we can gain from the resulting SFMEA table. First, the priority assigned to each hazard can lead the further SFTA which will explore the causes of the high priority hazards. Second, the identified causes of the hazards can aid in deriving hazards for the next level view by pointing out the possible components of the system which may lead to the hazards. Third, the right-most column which presents the trial hazards preventions is a possible input to lead to the further software architectural decision making and evaluation.

After the high priority hazards have been identified by the SFMEA analysis, the next step is to apply SFTA on them and trace back to find the requirement problems which may cause the hazards. Furthermore, scenarios can be derived based on the SFTA results.

### 7.2.2.3 Apply SFTA to identify causes of hazards

On the basis of the identified hazards by SFMEA, SFTA can work backward to find out the potential requirement problems which will contribute to the occurrence of the hazards. SFTA uses
Boolean logic to break down an undesirable event or situation (the root hazard) into the preconditions that could lead to the hazard.

We will use SFTA to trace back the causes of hazards within the according view. Taking the high priority hazards from the SFMEA table, the fault tree analysis will be carried out within the context of this view. Thus, for each view, its SFTA analysis will only trace back down to the requirement statements which are at the same detail level as this view. However, the same SFTA may be taken a step further in the next view if necessary.

We illustrate the SFTA analysis of the module view by analyzing the hazard ---- the “insulin overdose” and the according fault tree constructed as shown in Figure 26. From the figure, we can see that the root hazard has been traced until the module level requirement problems which may cause the root hazard have been found. Thus, the hazard prevention in the current level of view can be defined by cutting out all the paths leading to the root hazard. However the same root hazard may be further analyzed and decomposed into next level view if necessary, such that, the leave nodes in the fault tree of the upper level view may become the high-level nodes of the fault tree for the same hazard in the lower-level view.

If the SFMEA gives a static way to reveal the potential hazards of the system within a specific view, the SFTA makes it possible to unveil the behavioral characteristics of the system. The path in a fault tree defines a sequence of steps which will lead to the occurrence of the hazard. The leaf nodes will be the stimuli of this behavioral action. After taking step by step along the path, the system will respond by the root hazard happening. Thus, scenarios can readily be derived by following the paths in the fault tree.
7.2.2.4 Derive misuse scenarios from paths of the fault tree

A scenario describes the behavior of the system triggered by a stimulus. After a series of actions aroused by this stimulus, the system will give a specific response. Thus, scenarios describe those of the system behaviors. However, in the context of our approach, our analysis methods are to discover those system behaviors which should not be provided or occur in the system. We define those scenarios as misuse scenarios [26] which we are trying to prevent.

In the above section, we have seen the construction of the fault tree. The paths of fault tree are the behavioral tracks which will lead to hazards. Thus, to assure the safety quality of the system, those behavioral tracks should be eliminated during the development of the system.

From the definition of the misuse scenarios and the fault tree paths, we can find a close match between them. Therefore, deriving scenarios from the paths of the fault tree seems feasible and possible. The derived scenarios will be grouped under the root hazard. Thus, every scenario describes a possible system behavior sequence which is aroused by the stimulus (the leaf nodes of the fault tree) and results in the system response (the root hazard).
Taking the fault tree in Figure 26 as an example, we illustrate the derived misuse scenarios from the paths in the fault tree in the following Table 5.

Table 5. The derived misuse scenarios

<table>
<thead>
<tr>
<th>Hazard</th>
<th>Misuse scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insulin overdose administered</td>
<td>SM1: the sensor failure causes incorrect sugar level to be measured; the system’s response is to administer an overdose of insulin.</td>
</tr>
<tr>
<td></td>
<td>SM2: The sugar computation error causes incorrect sugar level to be measured; the system’s response is to administer an overdosed of insulin.</td>
</tr>
<tr>
<td></td>
<td>SM3: The timer failure cause correct dose to be delivered at the wrong time, thus resulting in an overdose of insulin to be delivered</td>
</tr>
<tr>
<td></td>
<td>SM4: The insulin computation error causes delivery system failure; the system’s response is to administer an overdosed of insulin.</td>
</tr>
<tr>
<td></td>
<td>SM5: The pump incorrect signals cause delivery system failure; the system’s response is to administer an overdosed of insulin</td>
</tr>
</tbody>
</table>

From the above derived misuse scenarios, we find that some of them do not fall into the software category. Thus, during the system development, software engineers can choose not to put effort into those non-software misuse scenarios while leaving them to hardware or system engineers.

The process of deriving misuse scenarios which we have presented here has several advantages over other derivation mechanisms based on the brainstorming session of experts. First, it has close relationship with safety analysis method, thus, the derived scenarios will directly reflect the safety requirements of the system. Second, it provides relative completeness. Because the misuse scenarios are derived from the paths of fault tree, there exist clear mappings from fault tree paths to the misuse scenarios. Thus, each possibility of the occurrence of the hazard will be considered. Third, it provides traceability. The misuse scenarios are grouped under the root hazard which can be traced back and forth to the earlier or later view, respectively. Thus, the origin and the refinement of a misuse scenario can be traced by following the SFMEA table and the SFTA for each view.

After the misuse scenarios have been derived, they can be input into the utility tree. Thus the part of the utility tree which is related to this specific architectural view can be completed.
7.2.2.5 Construct the utility tree

The utility tree will be another reusable asset of the approach besides the SFMTA tables. The constructed utility tree can be used to illustrate how the overall safety factors are instantiated into system behaviors. It can also be reused by becoming input to future architectural evaluation process which may evaluate the architectural decisions against these misuse scenarios.

As we have mentioned in the global analysis section, the utility tree will start at the root node (safety) and take the overall safety factors as the first-level nodes. The overall safety factors have been instantiated into the context of a specific view by applying SFMEA and SFTA analysis. The hazards derived by SFMEA have been prioritized. The SFTA will further refine the high-priority hazards into a series of system behaviors by mapping the paths of the fault tree into misuse scenarios. The lower-level nodes in the utility tree will be the groups of the misuse scenarios under the according factors. Figure 27 shows the constructed utility tree of the insulin delivery system after performing the activities in the module view.

Figure 27. The utility tree in the module view

The above utility tree reveals that if those misuse scenarios can be prevented from happening, the hazards will be prevented and thus the safety factors can be satisfied.
In this section, we have tried the detailed steps of the approach in the conceptual and module views of the insulin delivery system. The same technique is performed in the other two views, the code and the execution views. From the approach, we can see that it provides a systematic way to record the analysis results for each of the four architectural views. The resulting SFMEA tables and the utility tree can be used to (1) as data warehouses to locate a specific analysis result (a hazard or a scenario) and trace it back to its original view; (2) as bridges to connect the overall safety factors to the hazards, and finally to the misuse scenarios; and (3) as an input to future architectural evaluation where architectural decision may be evaluated by checking how efficiently the decisions are in terms of eliminating the misuse scenarios.

This chapter has presented a technique to identify, refine and enforce the safety-related requirements of a system. It describes how the safety factors can be instantiated into hazards and hence, the misuse scenarios by using SFMEA, SFTA and constructing the utility tree. The goal of safety will be satisfied when those identified hazards and misuse scenarios can be prevented. The chapter explains the detailed steps of the approach to refine the safety requirements and to derive the misuse scenarios from the perspectives of the four system views. We also investigate the possibility of software architectural evaluation by using the results of the approach.
CONCLUSION

This paper presents two analysis techniques, one to reason about the safety of the product family software and one to provide a structured way to develop safety-related scenarios for future software architectural evaluation.

The Fault Contribution Tree Analysis (FCTA) technique in Chapter One supports the hazard analysis for product family software and identifies those features that may contribute to the occurrence of the hazards. The technique is designed for product family software and is reusable across the product family domain.

The construction of the safety-related scenarios in Chapter Two supports software architectural evaluation. The technique can identify safety-related requirements from four different engineering perspectives and derive safety-related scenarios from hazard analysis methods. The technique is a systematic approach and provides relative completeness, detail, guidance and traceability.

The results of the two chapters show some future directions which merit investigation. The first analysis technique described here may provide a useful measure of software architecture support for fault tolerance. The software architecture can be evaluated according to the degree to which an architectural style prevents occurrence of paths from the fault contribution tree leaf nodes to the root failure. The second analysis technique suggests that a software architectural style can be evaluated against the derived safety-related scenarios. A better architectural style can be selected based on how efficiently it can prevent the occurrences of those scenarios.

It is hoped that the extension of FCTA to product families and the systematic derivation of safety-related scenarios for software architectural evaluation will encourage improved software safety analysis and architectural design of high-criticality systems.
REFERENCES


