Chapter II

Deriving Safety-Related Scenarios to Support Architecture Evaluation

Dingding Lu
Iowa State University, USA

Robyn R. Lutz
Iowa State University, USA

Carl K. Chang
Iowa State University, USA

Abstract

This chapter introduces an analysis process that combines the different perspectives of system decomposition with hazard analysis methods to identify the safety-related use cases and scenarios. It argues that the derived safety-related use cases and scenarios, which are the detailed instantiations of system safety requirements, serve as input to future software architectural evaluation. Furthermore, by modeling the derived safety-related use cases and scenarios into UML (Unified Modeling Language) diagrams, the authors hope that visualization of system safety requirements will not only help to enrich the knowledge of system behaviors but also provide a reusable asset to support system development and evolution.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Introduction

This chapter defines a technique to identify and refine the safety-related requirements of a system that may constrain the software architecture. The purpose of the approach presented here is to provide a relatively complete set of scenarios that can be used as a reusable asset in software architectural evaluation and software evolution for safety-critical systems.

For this purpose, we will identify, refine, and prioritize the safety-related requirements in terms of scenarios by using safety analysis methods. The resulting scenarios can serve as input to software architectural evaluation. By evaluating various kinds of architectural decisions against the input safety-related requirements, the evaluation approach will assist in the selection of an architecture that supports the system safety. The resulting scenarios are reusable during software evolution. By reusing those common scenarios, and hence the common architectural decisions, the cost of development and the time to market can be reduced.

The objective of this chapter is to introduce a technique that:

(1) Identifies and refines the safety-related requirements that must be satisfied in every design and development step of the system,

(2) instantiates the nonfunctional requirement – safety – into misuse cases and misuse scenarios that are further modeled by UML, and

(3) provides a reusable asset – utility tree – that may either support the engineering decision making during software development or become input to future software architectural evaluation.

Background

Currently many safety-critical systems are being built. Some safety-critical systems include software that can directly or indirectly contribute to the occurrence of a hazardous system state (Leveson, 1995). Therefore, safety is a property that must be satisfied in the entire lifetime of safety-critical systems.

Though safety is the key property of the safety-critical systems, some aspects of the system are not related to safety. A software requirement can be categorized as a safety-related requirement if the software controls or contributes to hazards (Leveson, 1995).

Identifying those safety-related requirements can guide the engineers to explore the most critical parts of the system and allocate development resources efficiently.

Two existing software safety analysis methods are adapted in this chapter to identify and prioritize the safety-related requirements for the targeted system (Lu, 2003). One is One is Software Failure Mode and Effect Analysis (SFMEA). SFMEA is an extension of hardware Failure Mode and Effect Analysis (FMEA) which has been standardized

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

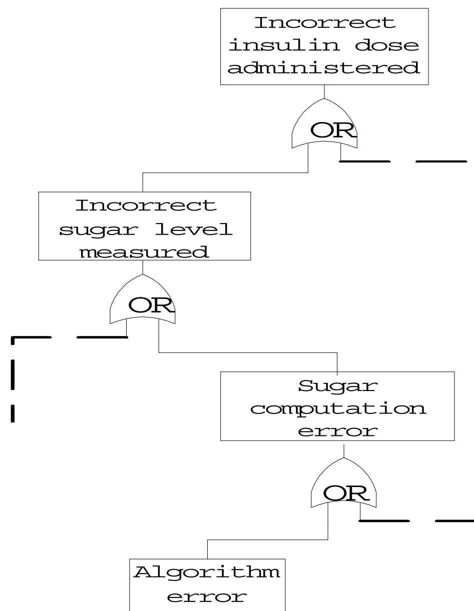
Table 1. The entries of SFMEA

Item	***Failure mode***	***Cause of failure***	***Possible effect***	***Priority level***	***Hazard reduction mechanism***

(Military Standard, 1980). SFMEA is well documented (Reifer, 1979; Lutz & Woodhouse, 1997) though there is no standard to guide performing SFMEA. SFMEA works forward to identify the "cause-effect" relationships. A group of failure modes (causes) are considered and the possible consequences (effects) are assessed. Based on the severity of the effect, the possible hazards are identified and prioritized. SFMEA uses tabular format. Some typical entries of SFMEA are depicted in Table 1.

The other is Software Fault Tree Analysis (SFTA). 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 (Raheja, 1991), but good results have been obtained in recent years by applying the technique to software systems as well (Hansen, Ravn & Stavridou, 1998; Leveson, 1995; Lutz, Helmer, Moseman, Statezni & Tockey, 1998; Lutz & Woodhouse, 1997; Lu & Lutz, 2002). SFTA uses Boolean logic to break down an undesirable event or situation (the root hazard) 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 hazard to the possible combinations of events or conditions that could lead to the occurrence of root hazard.

Figure 1. A sample software fault tree (Sommerville, 2001)



Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

SFTA uses a tree structure. The hazard is at the root of the tree and the intermediate nodes or leaves represent potential causes of the hazard. A sample software fault tree is displayed in Figure 1.

Figure 1 describes a portion of SFTA for a safety-critical system: the insulin delivery system (Sommerville, 2001), which monitors the blood glucose level of diabetics and automatically injects insulin as required. From the SFTA, the root hazard is the "incorrect insulin dose is administered". The events "incorrect sugar level measured", "sugar computation error", and "algorithm error" could be the causes of the root hazard and thus become the intermediate nodes or leaves of the SFTA.

As an important aspect of software development, software architecture will greatly influence the qualities of the system such as safety and reliability. Thus, the available software architectural styles must be evaluated against the safety requirements so that the best-fit architectural decisions can be made to build the system.

We here use the definition proposed by Bass, Clements, and Kazman (2003): "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." Software architectural styles such as client-server and pipe-filter have been defined by Shaw and Garlan (1996) to describe the component and connector types, and the set of constraints (e.g., performance, safety, and reliability) on how they can be combined. Definition of software architecture is important in terms of opening a channel for communication among different stakeholders, capturing the specific system properties in the early design stage, and providing architecture reuse for similar systems in the same domain.

When there is more than one architectural style that can be chosen for a system or a component of a system, architectural evaluation must be performed to trade off advantages and disadvantages among a variety of architectural decisions and to select the most suitable one for the system. Software architectural evaluation has been practiced since the early 1990's. The Architecture Tradeoff Analysis Method (ATAM) (Clements, Kazman, & Klein, 2001) is one such evaluation method 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 (Jones & Lattanze, 2001)

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 are 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 is further refined into scenarios that are the lower-level nodes in the utility tree. The scenarios are derived by brainstorming of experts and prioritized according to how much they will impact the system development.

The derived and prioritized scenarios are the inputs to the architectural evaluation step in ATAM. All the available architectural decisions are evaluated against those scenarios by assessing the potential effect of a specific architectural decision on each scenario (Lutz & Gannod, 2003). The effect is categorized into four kinds (risk, sensitivity, tradeoff, and non-risk) from the most critical to the least significant. Thus an architectural decision

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

which incurs risk on many of the scenarios may be abandoned and those which accommodate most of the scenarios will be selected.

From the ATAM process, we can see that the accuracy of the evaluation relies on the completeness of the input scenario set. However, the scenarios derived either from the experience checklists or from the experts' brainstorming may cause lack of coverage, guidance, detail, and traceability.

In order to avoid these problems, the complete view of a system is developed to decompose the system from different architectural perspectives.

The complete view of a system refers to the *4+1* view model of software architecture developed by Kruchten (1995). 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 that represent the behavioral instances of the software functionalities.

Hofmeister, Nord and Soni (1999) 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. 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, and so forth. The execution view deals with dynamic interactions between the software and the hardware or system environment. Each view deals with different engineering concerns. The safety-related requirements are mapped from the upper-level view to the lower-level view and are further refined in the context of that view. Segregating the system into four views gives a systematic way of revealing the system structure that helps to understand the domain in the early design stage, to identify the safety-related requirements, and to achieve relative completeness in scenario derivation.

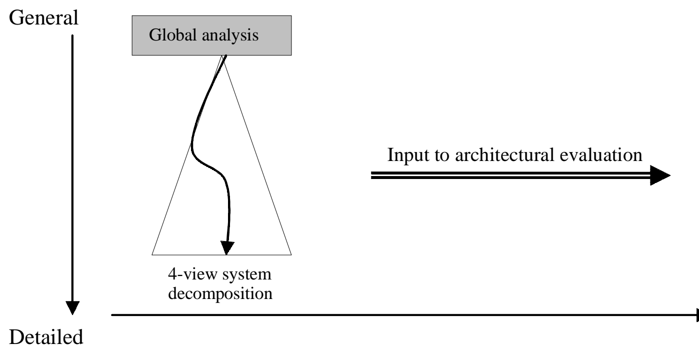
Overview of the Safety-related Scenario Derivation Process

In order to achieve improved coverage, completeness, and traceability in deriving safety-related scenarios, our process is founded on three key bases that are introduced in the previous section. First, the four architectural views of the system provide a way to study the system from four different perspectives and to consider different engineering concerns in each view. Second, the safety-related requirements define the overall safety factors of the system and guide the engineers to further explore those factors. Third, the safety analysis methods (SFMEA and SFTA) elicit and prioritize misuse cases and misuse scenarios (Alexander, 2003). The derived misuse cases and misuse scenarios transform the system safety into testable requirements that can serve as the input to the later architectural evaluation. The evaluation selects the suitable architectural decisions which will in turn impact the safety of the system.

Figure 2 depicts an overview of the process. The vertical direction describes the decomposition of the system into four views and maps the safety-related requirements

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2. The overview of the process



into each view. Both the knowledge of the system and the safety-related requirements are detailed and enriched as one proceeds in this direction.

There are three steps in the vertical direction, and the first step is global analysis. The objective of global analysis is to identify the overall safety factors that will be further refined or decomposed into the four architectural views. During the refining and decomposing, the undesirable system functionalities that may affect system safety are captured and represented as misuse cases (Alexander, 2003). For each misuse case, a set of misuse scenarios is derived to model the detailed system behaviors within the specific misuse case. Thus, the misuse scenarios are the instantiations of the overall safety factors.

The second step in the vertical direction is the four-view system decomposition. The four views consider the engineering concerns of the system from four different perspectives. The first two views (conceptual and module) include design issues that involve the overall association of system capability with components and modules as well as interconnections between them. The third view (code) includes design issues that involve algorithms and data structures. The fourth view (execution) includes the design issues that are related to hardware properties and system environment such as memory maps and data layouts.

Though most of the engineering concerns of each view are independent of other views, some may have impacts on the later views. On the other hand, the later views may place constraints back on the earlier views. Thus the relationship between the two views is bidirectional and iterative as shown in Figure 3.

The resulting overall safety factors from the global analysis are localized (refined and decomposed) within the context of each view in terms of misuse cases and misuse scenarios. Meanwhile, additional safety concerns may also be investigated and identified.

After the misuse cases and misuse scenarios are derived, the last step of the vertical direction can be performed; that is, to produce a utility tree with the *utility—safety* as the root node and the overall safety factors as the second-level nodes. The remaining

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3. The four views system decomposition

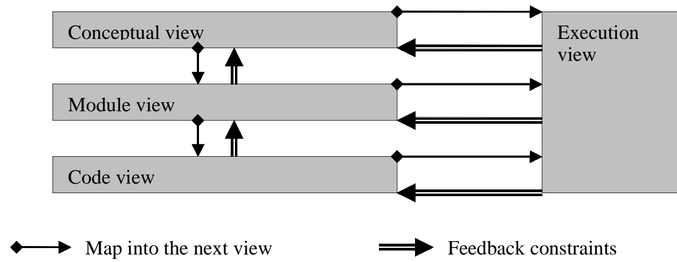
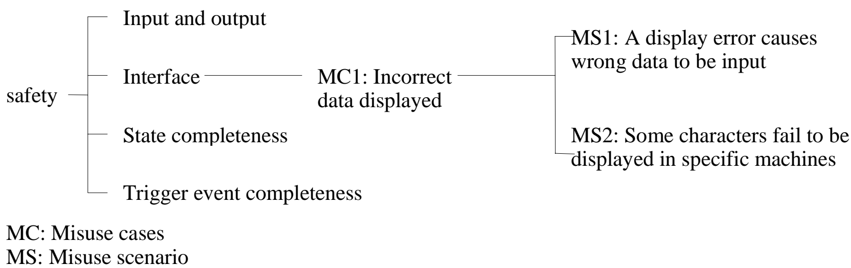


Figure 4. A sample utility tree



lower-level nodes are the derived misuse cases that are further decomposed into a set of misuse scenarios. The utility tree provides a top-down mechanism for directly translating the safety goal into concrete misuse scenarios. A sample utility tree is shown in Figure 4.

The advantages of constructing a utility tree are to provide a straightforward way to gather the main results of each step in the process and to serve as a reusable asset in later UML modeling by directly mapping the misuse cases and misuse scenarios into UML diagrams.

In the vertical direction, we have mentioned that misuse scenarios will be derived for each view to instantiate the safety-related requirements. How the misuse scenarios are derived and prioritized is the task that will be fulfilled in the horizontal direction.

There are two steps in the horizontal direction that will be performed for each view. The first step is the forward hazard analysis - the SFMEA where the misuse cases are to be identified. As explained before, SFMEA works forward from the failure modes to possible effects to identify and prioritize the potential hazards. The overall safety factors resulting from the global analysis will guide the hazard identification here.

Within each view, every one of the overall safety factors is considered in order to derive the related possible hazards. 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 to be further investigated. Thus, a traceable link among the four views is established

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

by looking at the recorded hazards in the SFMEA tables. The identified hazards are those undesirable functionalities of the system and thus are categorized as the misuse cases.

The second step is the backward hazard analysis: the SFTA where the misuse scenarios are to be derived. As described before, SFTA works backward to reason about the existence of a specific combination of basic events (the leaf nodes) 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 construct software fault trees.

A minimal cut set of a fault tree represents a combination of basic events that will cause the top hazard and cannot be reduced in number; that is, removing any of the basic events in the set will prevent the occurrence of the top hazard (Leveson, 1995). Every minimal cut set in a software fault tree will be mapped into a misuse scenario that consists of three components: stimuli, system reactions, and response. The basic events in the minimal cut set represent the stimuli of the misuse scenario. The paths from basic events to the root node through the intermediate nodes in the fault tree are the series of system reactions after the stimuli is triggered. The root hazard is the final response. For example, considering a minimal cut set of the fault tree in Figure 1, a misuse scenario can be derived: an algorithm error happens during a sugar computation, thus causing an incorrect sugar level measured; the system response is to administer an incorrect insulin dose to patient.

Combining the results of the SFMEA and SFTA analysis, every misuse case (the hazard) is decomposed into a set of misuse scenarios. The derived misuse cases and misuse scenarios become the lower-level nodes in the utility tree and are ready to be further transformed into UML diagrams.

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

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 that 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 misuse scenarios by mapping each minimal cut set of a fault tree into a scenario.
6. Repeat steps 2 to 5 until each of the four views is analyzed.
7. Construct a utility tree with the *safety* as root node. The second-level nodes are the overall safety factors identified in step 1, global analysis. The lower-level nodes are derived misuse cases (hazards) and misuse scenarios from each view.
8. Model the misuse cases and misuse scenarios by UML diagrams.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 5. The outline of the process

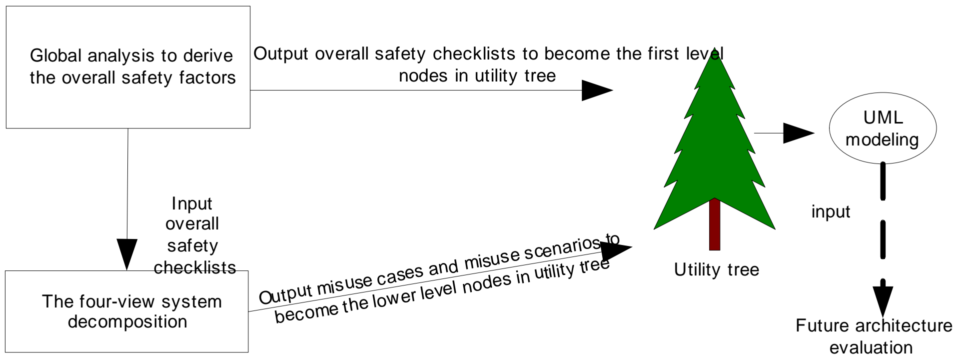
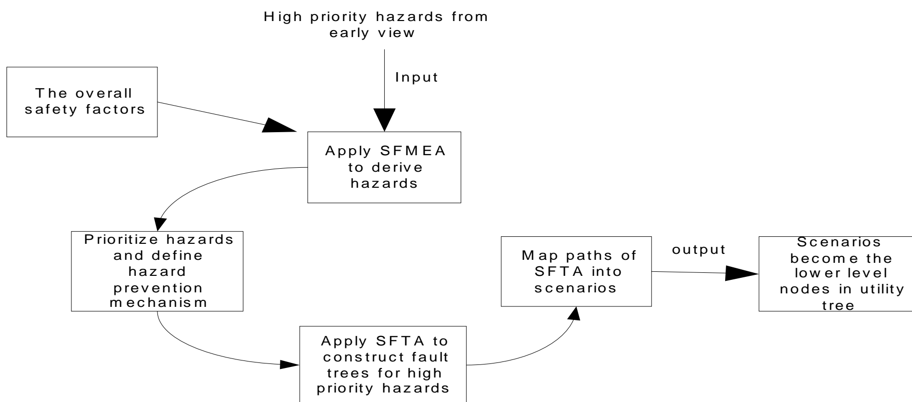


Figure 6. The activities within each view



We illustrate these steps of the overall process in the following two figures. Figure 5 outlines the overall process. Step 1 of the algorithm, the global analysis, will be performed before the four-view system decomposition starts. The identified overall safety factors will become the first-level nodes of the utility tree and will be mapped into each of the four views. After all the analysis activities (steps 2 through 5) within each of the four views are taken, the output misuse cases and misuse scenarios become the lower-level nodes in the utility tree. The resulting utility tree thus includes the main results of the process.

Figure 6 shows the activities involved within each of the four views. Those activities are involved in steps 2 through 6 of the algorithm. The overall safety factors will become the input to each of the four views together with the high-priority hazards of the earlier view.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

(But note that there will not be any high-priority hazards input to the conceptual view because it is the first one of the four views.)

The contributions of the technique we have presented here are (1) to provide a structured and systematic way to instantiate safety-related requirements into detailed and testable misuse scenarios by decomposing the system from the four architectural perspectives; (2) to identify the safety-related requirements by deriving the hazards in SFMEA and defining the prevention mechanisms for the hazards; (3) to adapt UML modeling by transforming the misuse cases and misuse scenarios into UML diagrams; and (4) to support future software architectural evolution by providing a detailed scenario-based 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 is the insulin delivery system (Sommerville, 2001). We selected this system based on its suitable size and its safety concerns.

Global Analysis

The global analysis is to identify the overall safety factors which may affect 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 and prevented during the testing stage of system development. To avoid either expensive rework, or the even more costly recovery from catastrophic consequences, these There are three activities in the global analysis: Identify and describe the overall safety factors, extract the subset of the safety factors based on the needs of a specific system, and evaluate the impact of the safety factors.

Identify and Describe the Overall Safety Factors

There are two ways to perform the identification of factors. 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 when a similar new system is going to be built. First, the valuable prior experience can be reused and part of the rework can be saved. Second, those same problems that happened and prevented development in the previous project can be easily avoided during 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. Particularly when a new system is going to be built, there may not be any previous 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 (Hofmeister, Nord, & Soni, 1999). Directly introducing the safety category to derive the overall safety factors brings two benefits. First, the safety category itself is clearly

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

described and organized; thus it provides a starting point to think about the safety factors of a specific system. Second, the safety category represents the general safety concerns and thus can be easily adapted to different kinds of systems.

In our process, the four architectural views safety category will be taken as the primary way to identify the overall safety factors. The accumulated experience checklist will work as the complement to ensure coverage as complete as possible.

Extract the Subset of the Safety Factors Based on the Needs of a Specific System

The resulting set of overall safety factors are relatively complete and general so they can 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 a keyboard or be otherwise manipulated by a human operator. Thus, the *human computer interface* safety factor will not be applicable in this case.

Depending upon the characteristics of a specific system, we will only focus on the subset of the identified safety factors that are applicable to the system. In this way, we scale down the overall number of the safety factors that need to be further analyzed.

Evaluate the Impact of the Safety Factors

The potential impact of a safety factor on the system development needs to be evaluated. The evaluation 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 severity of its possible effect. The higher the impact level is, the higher priority the factor will have.

Table 2. The safety factors

Software	***Management***	***Hardware***
Human-computer interface	Product cost	General-purpose hardware (processor, network, memory, disk)
Input and output variable	**Staff skills**	**Equipment quality**
Input and output Trigger	**Development schedule**	
Output specification	**Development budget**	
Output to trigger event relationships		
Specification of transitions between states		
Performance		

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Table 3. The impact levels of the overall safety factors

Safety factor	***Impact level***
Human-computer interface	High
Input and output variable	High
Trigger event	High
Output to trigger event relationships	*High*
Specification of transitions between states	*High*
Staffs skills	*Medium*
Development schedule	*Minor*
Development budget	*Minor*
Equipments quality	*High*

After the overall safety factors are identified and their priorities are assessed, they will become the first-level node in the utility tree and be ordered from the highest to the lowest priority.

We illustrate these three activities below by applying them to the insulin delivery system (Sommerville, 2001). The insulin delivery system uses sensors to monitor the patient's blood sugar level then calculates the amount of insulin needing to be delivered. The controller sends the command to the pump to pump the set amount of insulin. Finally, the insulin will be sent to the needle and injected into the patient's body. Thus, the insulin delivery system is an embedded system and uses several pieces of equipments to fulfill the task.

In step one, we import the four architectural views safety category defined by Hofmeister, Nord, and Soni (1999) and use the checklist of an advanced imaging solution system, IS2000 (Hofmeister et al.) as the complement. A set of safety factors is derived as in Table 2.

In step two, we identify the subset of the derived safety factors that are relevant to the characteristics of the insulin delivery system. The results are those safety factors that are marked as bold in Table 2.

In step three, we will analyze the possible impacts of the given safety factors on the system. Those software factors will interact with each other. If any of them has problems, the problems may propagate to other factors. Thus, we assign the impacts of the software factors as high. Among the management factors, the staff's skills may have an effect on 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 equipment, such as sensors and needle, will be critical and 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, as shown in Table 3.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

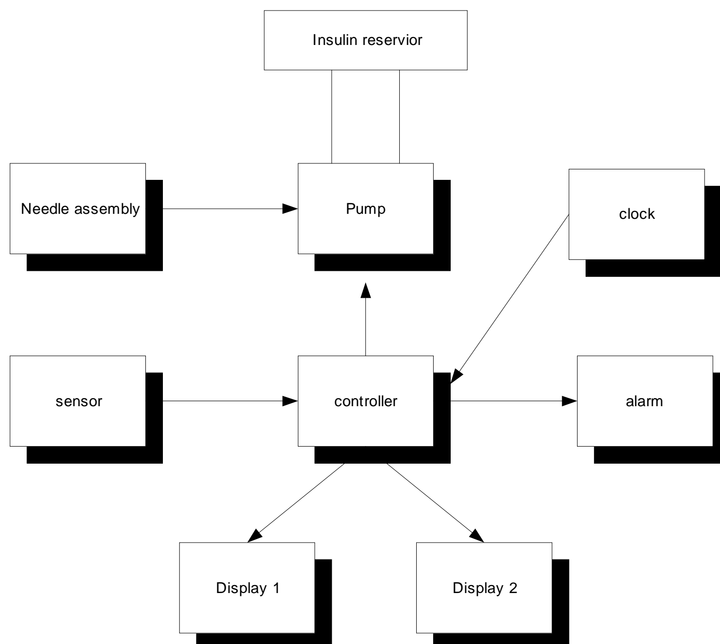
The Four-View System Decomposition

The Definition of the Four Views (Hofmeister, Nord, & Soni, 1999)

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 regards the functionalities of the system as black boxes and as being relatively independent from detailed software and hardware techniques. The functionalities are represented as *components* in the conceptual view. The communications and data exchanges between those functionalities are handled by *connectors*.

The *module view* takes a closer look into system 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 subcomponents and subsystems 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 subsystems or subcomponents will be described in the module view.

Figure 7. The conceptual view of the insulin delivery system (Sommerville, 2001)



Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

The *execution view* describes the system development in terms of its runtime platform elements which include software and hardware properties, such as memory usage, processes, threads, and so forth. It captures how these platform elements are assigned and how the resources are allocated to fulfill the system functionalities. The subsystems and subcomponents of the module view are mapped into the execution view by allocating the software or hardware resources for them.

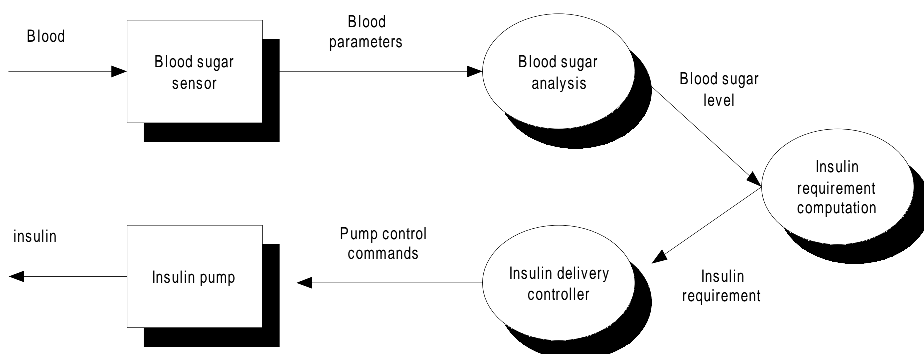
The *code view* describes how the system is implemented. In this view, all the functionalities of the system are fulfilled and all the runtime entities assigned by the execution view are instantiated by using a specific programming language. Thus the dependencies among system functionalities, library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

The conceptual view of the insulin delivery system is shown in Figure 7.

Figure 7 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 are explained as follows (Sommerville, 2001):

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 insulin dose from the controller, pumps the requested amount of insulin from the reservoir and sends it to the needle assembly.
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 that provides the controller correct current time.

Figure 8. Portion of the module view of the insulin delivery system (Sommerville, 2001)



Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

In Figure 8, 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 components, revealing the internal functionalities included in the components, and depicting the message passing among functional parts, a more detailed view of the system is provided.

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 tasks of the approach. There are two main activities that need to be fulfilled within each of the four views: hazard derivation and priorization using SFMEA, and hazard analysis and misuse scenarios derivation using SFTA.

Hazard Derivation and Prioritization – 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. Hazard analysis for the overall safety factors need to be performed to ensure the system safety. 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 and prioritizing.

Identify the Hazards and Their Causes

There are two ways to identify 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 refined within the detail level of the specific view. For example, when the *input and output* of the overall safety factors is mapped into the conceptual view of the insulin delivery system, the hazard *insulin dose is misadministered* can 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 have severe consequences. Thus, more attention and more resources need to be given to this hazard to prevent it from happening. The later view will take the high-priority hazards in the earlier view as input and further analyze and refine those hazards. 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. The conceptual view is the earliest view, thus no high priority hazards of an earlier view exist to be used as the input. Take the above hazard *insulin does is misadministered* of the conceptual view as an example. This hazard is classified as high priority and input into

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

the module view. In the module view, this hazard is refined into two hazards. 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 then the code view, to be further analyzed.

After the hazards have been identified, the possible reasons that may cause the hazards are assessed. The causes of the hazards provide valuable information for locating the possible erroneous requirements.

Prioritize the Hazards and Define Hazards Prevention Mechanism

When a set of hazards is derived, the potential effects they may cause during system development will be assessed. Each hazard will be assigned a level of priority according to the severity of its effect. The more severe the effect a hazard may cause, the higher the priority it will have. High-priority hazards may cause severe consequence to the system and thus deserve more attention to be paid and more system resources to be spent in order to prevent them from occurring.

Within the context 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 taken care of when there are enough development resources available.

From the hazard identification and prioritization discussed above, we can see the hazards are the undesirable functionalities of the system that might have catastrophic consequences. Thus we categorize the hazards as misuse cases in the later utility tree construction.

We illustrate these two steps in the conceptual and module views of the insulin delivery system. The hazards in the conceptual level view of the insulin delivery system have been identified as follows by applying the SFMEA analysis. However, we omit the causes of the hazards and the hazard prevention mechanisms. The reason for doing so is to provide simplicity by not delving into too much detail as we introduce the approach while maintaining the necessary connection between steps of the approach.

1. Insulin dose misadministered: 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

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

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 equipment 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: Clinical problems

   Hazard level: High

   Priority: Medium

In the module view of the system, first, the identified hazards with high priorities in the conceptual view will be further refined and explored. Second, the overall safety factors will be mapped into the module view. The resulting SFMEA table is Table 4. From the SFMEA table, we can find that two hazards (the insulin misadministered and sensor problem) in the conceptual view have been refined and four additional hazards have been identified.

Hazard Analysis and Misuse Scenarios Derivation – SFTA

Those high-priority hazards identified by SFMEA may have severe effects on system development and thus need to be further analyzed by using Software Fault Tree Analysis (SFTA).

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

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

Item	***Failure mode***	***Cause of failure***	***Possible effect***	***Priority Level***	***Hazard prevention***
Insulin dose administered	Overdose	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.	Cause patient to die	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.
	Underdose	Same	Same	High	Same
Sensor	Incorrect data	1.sensor break down 2.incorrect data sensed	Cause wrong blood parameter s to be output	High	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
Blood sugar analysis equipment	Incorrect analysis results	1.The input blood parameters are incorrect 2. The method used to analyze blood sugar is incorrect	Cause wrong blood sugar analysis results	High	Recheck the input blood parameters by sensor, discard the data if it's not within the normal range. Comprehensively test the analysis method
Insulin amount computation	Incorrect amount computed	1.The input blood sugar level is incorrect 2.the computation algorithm is incorrect	Cause incorrect insulin amount computed	High	Set a normal range for input sugar level, sounds alarm when it's out of range. Comprehensively test the algorithm
Insulin delivery controller	Incorrect pump commands sent	1.The input insulin amount is incorrect 2. The delivery controller breaks down	Cause incorrect pump control command s sent	High	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	Compare the input commands with the past history record, sound an alarm when there is any exception. Rigid quality check for the equipment

Hazard Analysis and Fault Tree Construction

Each high-priority hazard in the SFMEA table 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,

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 9. The fault tree of "insulin overdose" in the module view (Sommerville, 2001)

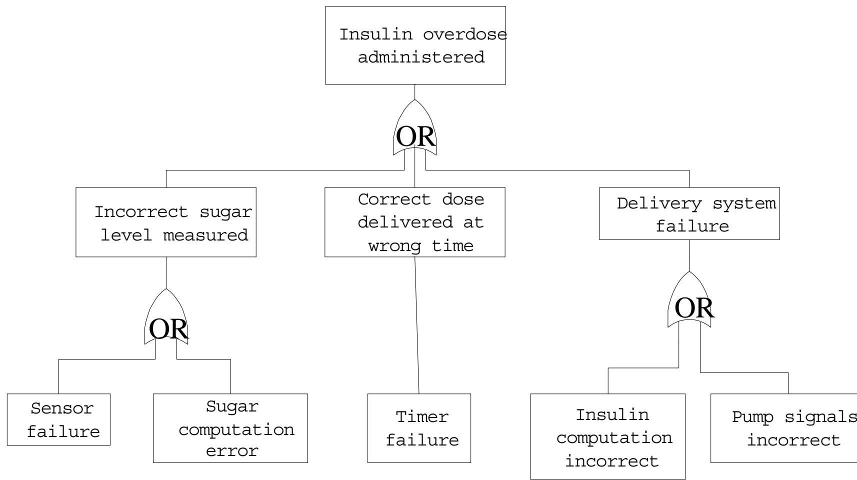


Table 5. The derived misuse scenarios

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

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.

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.

We illustrate the SFTA analysis of the module view by analyzing the hazard—the *insulin overdose*—and by constructing the associated fault tree as shown in Figure 9.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Misuse Scenarios Derivation

As 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 behaviors of system events that may cause the hazards. After the software fault trees are constructed, the misuse scenarios (Alexander, 2003) are ready to be derived and input into the utility tree.

The method used to derive those misuse scenarios is based on the minimal cut sets of the fault trees as we have discussed. Thus, every misuse scenario describes a possible system behavior sequence (intermediate nodes along the paths from the minimal cut set to the root hazard) which is aroused by the stimulus (the basic events of the minimal cut set) and results in the system response (the root hazard).

The misuse scenarios derived from the fault tree in Figure 9 are listed in Table 5.

Utility Tree Construction and UML Modeling

We have discussed the steps that need to be performed in our process. We are now ready to compose the main results and prepare for the further UML modeling.

Construct the Utility Tree

As we mentioned before, one of the advantages of constructing a utility tree is to provide a systematic way to gather the main results of the process. Hence, the hierarchical relationships among nodes in the utility tree clearly depict the traceable link for results from steps of the process.

We take the insulin delivery system as an example to construct the utility tree. The resulting utility tree is shown in Figure 10.

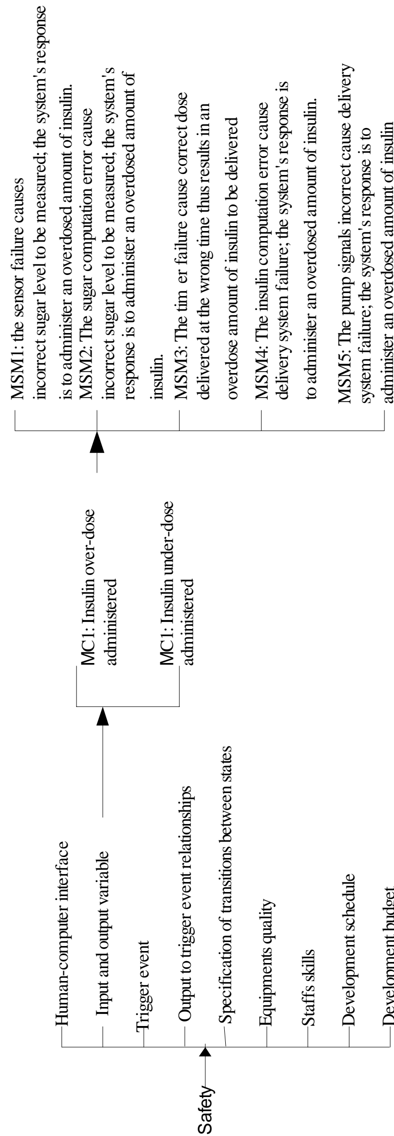
The root of the utility tree is *safety*, which is the key concern of the system development. The overall safety factors identified in the Global Analysis section are the second-level nodes. The misuse cases and misuse scenarios derived during SFMEA and SFTA analysis for each of the four views become the lower-level nodes grouped under corresponding safety factors.

UML Modeling

To utilize the standard modeling notation provided by UML, the misuse cases and misuse scenarios are transformed into UML use case diagrams and sequence diagrams. The original definitions of the UML use case diagram and sequence diagram are notations that represent the intended system functionalities and system behaviors. It is worth

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 10. The utility tree of the insulin delivery system



Dotted line describes the omitted part of the utility tree
MC1: MC - misuse cases, 1 - serial number
MSM1: MS - misuse scenarios, M - module view, 1 - serial number

noting that our misuse cases and misuse scenarios are those potentially hazardous system functionalities and behaviors that need to be prevented. The UML modeling for the misuse cases (MC1 and MC2) and misuse scenario (MSM1) of Figure 10 are illustrated in Figure 11 and Figure 12, respectively.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 11. The use case diagram of misuse cases UC1 and UC2

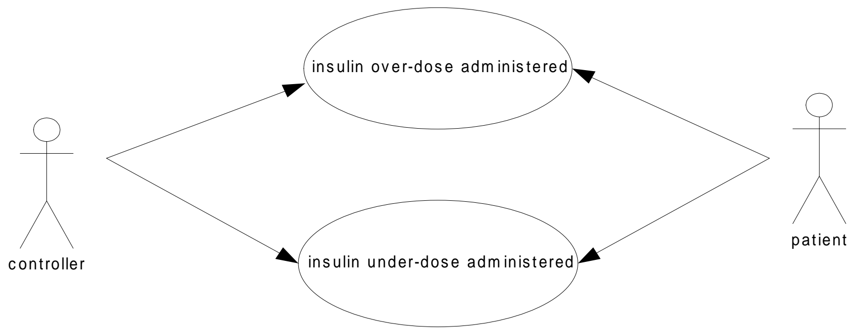
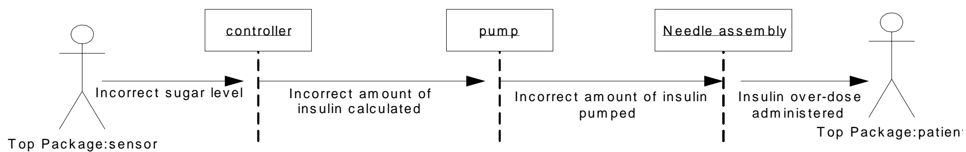


Figure 12. The sequence diagram of misuse scenario MSM1



There are several advantages of modeling the resulting misuse cases and misuse scenarios into UML diagrams.

First, the graphical tools provided by UML help us visualize those undesirable system functionalities and behaviors. The interrelationships among system components are depicted, such that appropriate architectural decisions can be made to prevent the occurrences of those undesirable functionalities and behaviors.

Second, UML is a standard modeling language. After modeling the misuse cases and misuse scenarios into UML diagrams, those existing UML analysis methods can be adapted to further investigate the prevention mechanisms for the misuse cases and misuse scenarios.

Third, future software architectural decisions can also be modeled in UML. Thus, the misuse cases and misuse scenarios can serve to derive the architectural constraints during UML modeling.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Discussion and Future Work

We have explained our analytical approach in detail by using a case study, and, after performing the entire approach, we have two main results: an SFMEA table for each of four views, and a utility tree.

The SFMEA Table

The analysis produces an SFMEA table for each of the four views. The resulting four SFMEA tables provide traceability to the origins of the identified 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 SFMEA tables play an important role in software evolution. The four SFMEA tables are reusable. The defined hazard-prevention mechanisms can be reused during the detailed system design stage. The common hazard-prevention mechanisms can be directly imported when the same hazards are encountered during the later software evolution. 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.

The Utility Tree

After the construction of the 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 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 an architectural decision can prevent or how efficient an architectural decision is in preventing misuse scenarios, a better decision can be selected.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

A possible direction for future work is to expand the approach to handle product line software. A product line is defined as a set of systems sharing a common, managed set of features satisfying a particular market segment or mission (Clements & Northrop, 2001). Product line software development supports reuse (building member systems from a common asset) and software evolution (a new member system is built by reusing common features of the existing member systems) (Clements & Northrop). By handling product line software, the approach can have improved adaptability during software evolution.

References

Alexander, I. (2003). Misuse case: Use cases with hostile intent. *IEEE Software, Jan/Feb,* 58-66.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice* (2nd ed.). Boston: Addison-Wesley.

Clements, P., Kazman, R., & Klein, M. (2001). *Evaluating software architectures: Methods and case studies*. Boston: Addison-Wesley.

Clements, P., & Northrop, L. (2001). *Software product lines: Practice and patterns*. Boston: Addison-Wesley.

Hansen, K.M., Ravn, A.P., & Stavridou, V. (1998). From safety analysis to software requirements. *IEEE Transactions on Software Engineering, 24(7),* 573-584.

Hofmeister, C., Nord, R., & Soni, D. (1999).*Applied software architecture.* Boston: Addison-Wesley.

Jones, LG., & Lattanze, A.J. (2001). *Using the architecture tradeoff analysis method to evaluate a war game simulation system: A case study*. Software Engineering Institute, Carnegie Mellon University, Technical Note CMU/SEI-2001-TN-022.

Kruchten, P. (1995, November). The '4+1' View Model of Software Architecture. *IEEE Software, 12(6),* 42-50.

Leveson, N.G. (1995). *Safeware: System safety and computers*. Boston: Addison-Wesley.

Lu, D. (2003). Two techniques for software safety analysis. Master's thesis, Iowa State University, Ames, IA.

Lu, D., & Lutz, RR. (2002). Fault Contribution Trees for Product Family. *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02), November 12-15*, Annapolis, MD.

Lutz, RR., & Gannod, G. (2003). Analysis of a software product line architecture: An experience report. *The Journal of Systems and Software, 66(3)*, 253-267.

Lutz, RR., Helmer, GG., Moseman, MM., Statezni, DE., & Tockey, SR. (1998). Safety analysis of requirements for a product family. *Proceedings of the Third International Conference on Requirements Engineering (ICRE '98),* 24-31.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Lutz, RR., & Woodhouse, RM.(1997).Requirements analysis using forward and backward search. *Annals of Software Engineering, 3*, 459-474.

Military Standard (1980). *Procedures for performing a failure mode, effects and criticality analysis, MIL-STD-1629A.*

Raheja, D. (1991). *Assurance technologies: Principles and practices.* New York: McGraw-Hill

Reifer, D.J. (1979). Software failure modes and effects analysis. *IEEE Transactions on Reliability, R-28, 3,* 247-249.

Shaw, M., & Garlan, D. (1996). *Software architecture, perspectives on an emerging discipline.* New York: Prentice Hall.

Sommerville, I. (2001). *Software Engineering* (6th ed.). Harlow, UK: Addison-Wesley.

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.