PLFaultCAT: A Product-Line Software Fault Tree Analysis Tool[*]

Josh Dehlinger dehlinge@cs.iastate.edu (http://www.cs.iastate.edu/~dehlinge)

Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA, 50011

Robyn R. Lutz rlutz@cs.iastate.edu (http://www.cs.iastate.edu/~rlutz)

Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA, 50011

and Jet Propulsion Laboratory/Caltech, Pasadena, CA, 91109

Abstract. Industry currently employs a product line approach to software development and deployment as a means to enhance quality while reducing development cost and time. This effort has created a climate where safety-critical software product lines are being developed without the full range of accompanying safety analysis tools available to software engineers. Software Fault Tree Analysis (SFTA) is a technique that has been used successfully to investigate contributing causes to potential hazards in safety-critical applications. This paper further extends the adaptation of SFTA to product lines of systems by describing a software safety analysis tool called PLFaultCAT. PLFaultCAT is an interactive, partially-automated support tool to aid software engineers in the application of product-line software SFTA. The paper describes the integration of product-line SFTA and PLFaultCAT with the software development life cycle. The description includes the initial construction of the product-line SFTA as well as the automated derivation of software fault trees for product line members. The technique and tool are illustrated with a small case study throughout the paper.

Keywords: product-line engineering, software fault tree analysis, safety analysis tools, hazard analysis

1. Introduction

Reusability has transformed entire industries and caused software engineers to adapt their methods to further this goal. The software product-line concept supports reuse by developing a suite of products sharing core commonalities (Clements, 2002). However, development of safety-critical software product lines in industry has emerged ahead of the development of product-line, safety analysis techniques and tools. This has created a lack of techniques and

[*] This paper is an extended version of the paper "Software Fault Tree Analysis for Product Lines" presented by the authors at HASE 2004, Tampa, FL. March 2004.

tools available to software engineers to ensure the safe reuse of software components throughout a product line (Lutz, 2000). It is only after a full suite of safety analysis tools and techniques are available to software engineers to ensure the safety in safety-critical product lines that safety-critical software product lines will gain organizational and industrial acceptance and assume more responsibility in everyday safety-critical applications.

This paper offers additional assurance to software engineers by providing a software safety analysis tool, called PLFaultCAT (**P**roduct-**L**ine **Fault** Tree **C**reation and **A**nalysis **T**ool). This tool builds on a previously developed technique that adopted Software Fault Tree Analysis (SFTA) to product line safety analysis (Dehlinger and Lutz, 2004). PLFaultCAT is an interactive, partially-automated software support application to aid software engineers with the visualization and pruning process of a product-line SFTA. Specifically, the tool exploits the reusability inherent in product-line engineering by deriving reusable safety analysis assets (i.e., the product-line members' fault trees) for future systems within the existing product line.

The product-line SFTA maintains the safety analysis qualities of traditional SFTA while accommodating reusability in product-line engineering. Traditional SFTA targets the safety analysis of potentially harmful states for a single product. The product-line SFTA, however, incorporates the variabilities among the different products and contributes to the safety analysis for the entire product line without performing traditional SFTA serially on each product-line member. A new SFTA for a product line member can be derived almost automatically with PLFaultCAT using its pruning algorithm. The aim of this technique and tool is to support the confident reduction of the safety analysis needed on a new product in the product line and, ultimately, a less expensive and shorter product development process.

The contribution of this paper is to further investigate how and to what extent the product-line SFTA technique, supported by the PLFaultCAT tool, can be used by software engineers as a reusable safety analysis. Our approach employs Ardis and Weiss' Family-Oriented Abstraction, Specification, and Translation (FAST) model (Ardis and Weiss, 1997). This model employs a two-phase software engineering approach: the domain engineering phase and the application engineering phase (Weiss and Lai, 1999). The domain engineering phase defines the product line and constructs the product-line SFTA with the aid of the PLFaultCAT tool; the application engineering phase develops and performs the safety analysis on new product-line members. We first provide a framework for the construction, aided by PLFaultCAT,

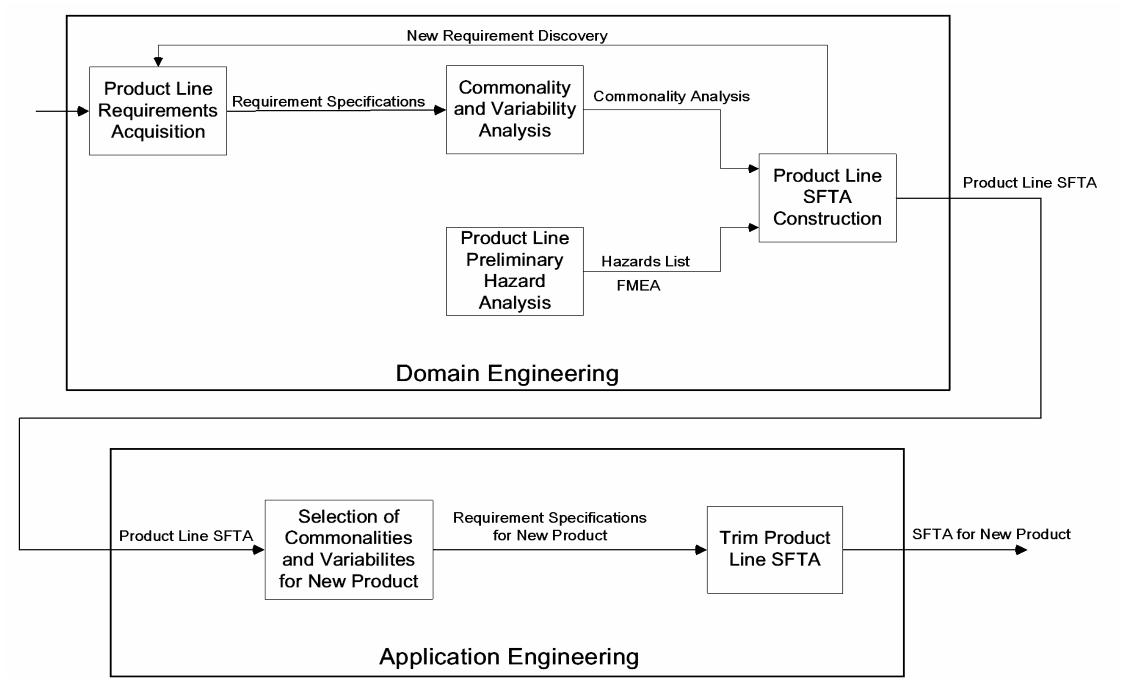

New Requirement Discovery
Product Line Requirements Acquisition
Requirement Specifications
Commonality and Variability Analysis
Commonality Analysis
Product Line SFTA Construction
Product Line SFTA
Product Line Preliminary Hazard Analysis
Hazards List
FMEA
Domain Engineering
Product Line SFTA
Selection of Commonalities and Variabilites for New Product
Requirement Specifications for New Product
Trim Product Line SFTA
SFTA for New Product
Application Engineering

Figure 1. An overview of the product-line SFTA technique

of a product-line SFTA during the domain engineering phase and then supply the means for reusing the product-line SFTA for new members as it is implemented in the PLFaultCAT tool. Within the application engineering phase we utilize PLFaultCAT to facilitate the derivation of new product-line members' fault tree(s).

Figure 1 provides an overview of this two-phased technique. The role of PLFaultCAT in this framework primarily resides in the application engineering phase. Although PLFaultCAT can assist in the initial graphical representation of a product-line fault tree, the chief contribution of the PLFaultCAT tool is to automatically produce the fault tree artifacts that software engineers desire at the end of the application engineering phase.

The remainder of the paper is organized as follows. Section 2 describes background information and related work in product line engineering and safety analysis. Section 3 presents an overview of PLFaultCAT and discusses its software architecture. Section 4 describes the construction of the product-line SFTA using PLFaultCAT for the Floating Weather Station (FWS) product line from (Ardis and Weiss, 1997; Weiss and Lai, 1999) to illustrate the approach and the use of PLFaultCAT. Section 5 discusses how to use the product-line SFTA to generate a product-line member's individual software fault tree(s). This section details the process of pruning a product-line SFTA to generate a single member's SFTA and illustrates how

PLFaultCAT can be employed to perform the pruning automatically. Section 6 presents an evaluation and analysis of the product-line SFTA technique using PLFaultCAT. Finally, Section 7 provides concluding remarks and future research directions.

2. Background and Related Work

The work presented here builds upon the increasingly overlapping areas of software safety analysis and software product-line development. This section discusses background information and related work in these areas of software engineering.

2.1 Safety-Critical Systems

Safety-critical systems can directly or indirectly compromise safety by placing a system into a hazardous state causing the potential loss or damage of life, property, information, mission, or environment (Leveson, 1995). Safety-critical software systems are being assimilated into our everyday lives in a vast range of domains and markets (Lutz, 2000). Safety-critical software runs applications such as pacemakers, aircraft flight-control systems, military weapons systems and nuclear power monitoring systems.

Just as autonomous software products have caused accidents, product-line software applications have also contributed to catastrophic losses. For example, the Therac-25 medical system and the Ariane 5 losses were accidents caused, in part, by product-line engineering mistakes (Leveson, 1995; Sommerville, 2004).

2.2 Safety Analysis Techniques

The following subsection describes two of the most common safety analysis techniques used by software engineers on safety-critical software: Software Fault Tree Analysis (SFTA) and Software Failure Modes and Effects Analysis (SFMEA). The first technique is the focus of this research. The second technique, when available, complements and aids in the development of a product-line SFTA.

2.2.1. SFTA. SFTA is a traditional safety analysis technique that has proven to be an essential tool for software engineers during the design phase of a safety-critical software product (Hansen, Ravn and Stavridou, 1998; Leveson, 1995; Lutz and Woodhouse, 1997; Pai and Dugan, 2002). SFTA is a top-down, backward search method utilizing Boolean logic to depict the causal event contributing to an undesirable event (the root node). The intermediate, causal event nodes combine with logic gates to describe how the parent node event can occur.

SFTA is a backward search analysis. Analysis begins at the root node with the engineer describing all possible paths to the root node through the event nodes and logic gates of the lower levels of the tree. For safety-critical systems, the root node of the tree will often represent a system-wide, catastrophic event taken from a preexisting hazards list (Leveson, 1995). The hazard represented by the root node is hypothesized to have occurred, and the engineer proceeds to determine the set of necessary preconditions causing the root node. The set of possible causes are joined to the parent node by standard logical relations represented via logic gates to describe their contributing relation. This process continues through each level of the constructed subtree until basic events are reached or until the level of subsystem detail is achieved.

Previous research in SFTA has represented multiple possible outcomes of a component failure, for example, depending on whether a warm spare is available (Coppit and Sullivan, 2003; Pai and Dugan, 2002). However, these describe single-system behavior rather than the product-line behavior of concern here.

2.2.2. SFMEA[1]. SFMEA is a tabular, forward-based search technique. Unlike SFTA, SFMEA is a bottom-up method since it starts with the failure of a component or subsystem and then looks at its effect on the overall system. SFMEA first lists all the components comprising a system and their associated failure modes. The effects on other components or subsystems are evaluated and listed along with the consequence on the system for each component's failure mode(s). Like SFTA, SFMEA is only as good as the domain and system expertise of the analyst.

[1] Software Failure Modes and Effects Analysis (SFMECA) is a similar safety analysis technique that additionally assigns a criticality rating to each failure mode.

2.3 Software Product-Line Engineering

A software product line is defined as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" (Clements and Northrop, 2002). The members of a particular product line differ from each other via a set of allowed variabilities/features.

The benefits of the product-line concept come from the reuse of the commonalities of the product line in the development of a new product-line member (Sommerville, 2004). Thus, the assets gained from the initial engineering of the product line, such as the underlying architecture, requirements, safety analyses and testing, can be at least partially applied to any new product-line member. In this sense, product line engineering allows for the amortization of costs in startup development and analysis of the initial product line members over the development of the entire product line. In fact, studies suggest that the product-line engineering concept can reduce the development and production time as well as the overall cost and increase the product quality by a factor of 10 times or more (Schmid and Verlage, 2002).

Requirements for product lines are often identified and specified through a Commonality and Variability Analysis (CA). The CA, as detailed by (Ardis and Weiss, 1997) and (Weiss and Lai, 1999), provides a comprehensive definition of the product line that details the commonalities, variabilities and inherent dependencies of the product line. This analysis technique aids in providing relevant domain definitions, the core set of product traits and the scope of the product line.

Commonalities describe requirements of the entire product line. Variabilities capture specific features not contained in every member of the product line. Variabilities also frequently have associated "parameters of variation" that detail the degree to which the variability can occur (Weiss and Lai, 1999). The parameters of variation describe the acceptable range of variation.

A variability's parameters of variation within a product line often fall into one of three categories: Boolean parameters of variation, enumerated parameters of variation, or range parameters of variation. These categories of parameters of variation get increasingly more difficult to analyze for safety as the complexity in the number of choices increases. Boolean parameters of variation are those variabilities that can either be present within a product-line member or not. An enumerated parameter of variation is any variability in which the product-line member must choose from a relatively small list of values for a particular variability. A simple example of an enumerated parameter of variation is "Widget X can either be blue, green, red, or yellow". A range parameter of variation are those variabilities in which the product-line member must have a precise number associated with the variability, where the number lies within the range of acceptable parameters of variation specified in a CA. For example, "Widget X may have between 1 and 100 user functions" is a simple range parameter of variation.

A product-line dependency restricts and/or dictates some combinations of variability subsets from being viable products in the form of "mutual exclusion" or "requires" variability dependencies (Kang et al., 1999; Doerr, 2002). A dependency requirement can thus take the form "Any product-line member that has *Variability A* can not also have *Variability B*" or in the form "Any product-line member that has *Variability C* must also have *Variability D*". This example indicates that any member of this product line is restricted from displaying both of these behaviors. Dependency requirements can derive from actual physical limits, undesired or infeasible combinations of behaviors, user restrictions, or business decisions. Building a new product in the product line (application engineering) entails selecting values for all the parameters of variation consistent with the dependencies as detailed in the CA. To aid in this, Padmanabhan and Lutz developed DECIMAL, a requirements validation tool to certify that a set of requirements for a proposed product line member does not breach the dependencies or constraints of the product line (Padmanabhan and Lutz, 2002).

Dependency requirements are especially important for the hazard analysis of a product line and should be explicitly documented. By reducing the subset of potential viable products stemming from the product-line definition, we reduce the scope of needed hazard analysis considerations. With product-line SFTA, such constraints often significantly reduce the number of subtrees.

Earlier work has investigated the application of various safety analysis techniques to product line requirements. In (Lutz, 2000), Lutz specifies a telescope subsystem as a product family and incorporates bi-directional safety analysis to identify additional requirements. Similarly, in (Lutz et al., 1998), Lutz et al. performed a forward and backward search for hazards on representative members of a flight instrumentation display product family in hopes of deriving additional safety requirements. The work described here advances Lu and Lutz's Fault

Contribution Tree Analysis (FCTA) (Lu and Lutz, 2002) for product lines by utilizing the more familiar fault tree methodology and by accommodating the use of commonality and variability values within the analysis and depiction of the product-line SFTA.

3. PLFaultCAT Overview and Software Architecture

This section introduces and briefly describes the PLFaultCAT tool. PLFaultCAT is the software tool developed to aid in both the domain engineering phase for initial product-line software fault tree analysis (SFTA) development and representation as well as in the application engineering phase for the derivation of product line members' software fault tree(s) from the product-line software fault tree(s) developed in the domain engineering phase. In this section, we present an overview of the PLFaultCAT tool in subsection 3.1 and give a description of the software architecture in subsection 3.2.

3.1 PLFaultCAT Overview

PLFaultCAT is a tool-assisted visualization and pruning application for the creation and analysis of product-line software fault trees. PLFaultCAT is an extension of the FaultCAT application (Burgess, 2003). FaultCAT is an open-source fault tree creation tool written in Java that is primarily geared towards analyzing a system for faults to determine how faults can affect other parts of the system (Burgess, 2003). FaultCAT does this by attaching fault probabilities to each node. FaultCAT provides a user the ability to graphically construct and represent the nodes and logic gates of a traditional fault tree. A complete discussion of the construction of a product-line software fault tree using PLFaultCAT is given in Section 4.

PLFaultCAT internally stores the fault trees in an XML format, making it easy to manipulate and alter. This is important because product lines routinely evolve, and the safety analysis must accordingly be updated. PLFaultCAT builds on the existing XML storage format of a fault tree in FaultCAT. During the pruning process of the application engineering phase, PLFaultCAT utilizes the XML DOM parser to perform the pruning necessary to generate a product-line member's fault tree(s) from the product-line SFTA. A full discussion of the pruning

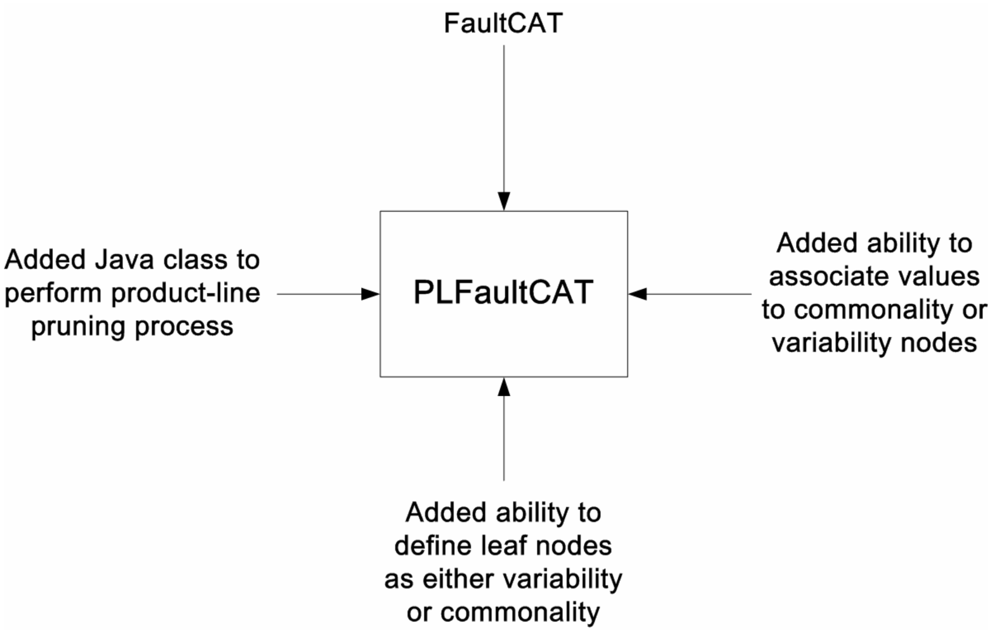

FaultCAT
Added Java class to perform product-line pruning process
PLFaultCAT
Added ability to associate values to commonality or variability nodes
Added ability to define leaf nodes as either variability or commonality

Figure 2. Software architecture of PLFaultCAT

algorithm and how it is handled in PLFaultCAT is given in Section 5. In addition to the graphical and XML view of the fault tree, PLFaultCAT presents a textual overview of a fault tree that lists the nodes of a fault tree, the type of a leaf node (either a commonality or a variability) and the value of a leaf node commonality or variability. Section 4.1, Step 3 describes these values in detail.

3.2 PLFaultCAT Software Architecture

The PLFaultCAT software architecture is built directly upon the software architecture of the original FaultCAT application. Thus, the majority of the PLFaultCAT tool inherits the base software architecture of FaultCAT. PLFaultCAT enhances FaultCAT by adding onto the software architecture the functionality needed to accommodate the creation and analysis of product-line software fault trees. Figure 2 shows the architecture of the PLFaultCAT application.

PLFaultCAT maintains all the functionality of FaultCAT and can still accommodate the creation and analysis of a single product software fault tree. To achieve this, the original FaultCAT software architecture, including the class structures, is maintained. Any additional functionality added to the already existing classes of FaultCAT has been tested to ensure that it does not interfere with FaultCAT's intended functionalities.

3.3 Implementation of PLFaultCAT

The major contribution to the PLFaultCAT tool is to add the nearly automatic pruning process of deriving a product-line member's fault tree from the product-line SFTA. Within PLFaultCAT, this is implemented as additional Java classes not found in FaultCAT. These Java classes provide the interactive, GUI-driven interface to allow a user to actively select the variabilities to include in any new product-line member. The selected variabilities then are used to properly prune the stored product-line SFTA to produce the derived product-line member's software fault tree.

To facilitate the creation of a product-line software fault tree, PLFaultCAT provides the ability to define a leaf node within a fault tree to be a fault associated with either a commonality requirement/component or a variability requirement/component. Defining leaf nodes as either being coupled to a commonality or a variability allows for the pruning process to determine which branches or subtrees are relevant for a given fault tree and a selected set of variabilities.

Lastly, PLFaultCAT provides the ability to specify the value(s) for a particular commonality or variability comprising the product line. Assigning the value(s) of a particular commonality or variability to a leaf node within a fault tree provides (1) an association of the leaf node with specifically what the choice of variability must be in order to contribute to its parent event node and its associated subtree and (2) a heuristic for the pruning algorithm to resolve those branches or subtrees that are applicable for a given fault tree and a selected set of variabilities and their values.

4. Domain Engineering - Constructing the Product-Line SFTA

The safety analysis for domain engineering of product lines uses results from the Preliminary Hazard Analysis (PHA). A PHA identifies the systems' hazards at an early stage of development with the aim of determining their impact on the system (Leveson, 1995). A domain hazards list will often exist prior to the development of the product line. If no preexisting hazards list is available, procedures exist to establish a workable, comprehensive list (Douglass, 1999). The creation of the hazards list requires extensive domain expertise and may be

Commonalities

C_1. Each FWS shall contain a computer, a set of wind and temperature sensors and a radio receiver and transmitter.
C_2. Each FWS shall have one or more wind sensors to obtain and record wind magnitude in knots and wind direction.
C_3. The FWS shall contain three different algorithms for calculating wind speed.
C_4. The wind speed value reported shall be calculated as the weighted averages calculated by different algorithms given the same series of inputs.
C_5. Each FWS shall broadcast wind, temperature and positional information every 60 seconds.
C_6. The wind sensors shall be read every 30 seconds.
C_7. Each FWS shall broadcast detailed weather reports in response to requests from passing vessels in the RAINFORM format.

Variabilities

V_1. A FWS may contain different types of sensors, such as wave spectra sensors.
V_2. A FWS may be equipped with an emergency switch such that, if a sailor is able to reach the buoy, a flipped switch would initiate SOS broadcasts from the buoy.
V_3. A FWS may report the wind magnitude in KPH or MPH in addition to in knots.
V_4. The broadcast data rate may vary.
V_5. For any FWS equipped with an emergency switch, a SOS signal broadcast request shall replace the ordinary 60-second message after a sailor flips the emergency switch. This should continue until a vessel sends a reset signal.
V_6. The type of vessel a FWS can respond to requests from may vary.

Dependencies

D_1. A FWS may only contain the capability to report the wind speed in units of KPH or MPH in addition to knots but not both KPH and MPH.
D_2. Any FWS with an emergency switch feature must also have the associated software to activate the SOS signal broadcast processes.

Figure 3. An excerpt of the FWS Commonality and Variability Analysis

performed in parallel with the Commonality and Variability Analysis (CA) described in Section 2.3. Alternatively, states from the "Possible Effects" column of the Software Failure Modes, Effects and Criticality Analysis (SFMECA) table can be used as a source of hazards for the root nodes of the product-line Software Fault Tree Analysis (SFTA) as they represent states that must be avoided.

Following the initial product line requirements acquisition in the FAST method, a precise definition of the members of a product line is achieved through the creation of a CA. Figure 3 provides a portion of the CA for a representative Floating Weather Station (FWS) from (Ardis

Parameter	Meaning	Domain	Binding Time	Default
WindSensorCount	Number of wind sensors	[1 .. 5]	Specification	1
WaveSpectraSensorCount	Number (if any) of wave spectra sesnsors	[0 .. 5]	Specification	0
EmergencySwitch	The presence of an emergency switch to activate an SOS signal transmission	[TRUE, FALSE]	Specification	FALSE
WindMagnitudeAdditionalUnits	Additional units of wind magnitude in which it is possible to transmit	[-, MPH, KPH, MPH + KPH]	Specification	None

Figure 4. An excerpt of the FWS Parameters of Variation

and Weiss, 1997; Weiss and Lai, 1999) that will be used as a running example to illustrate the activities involved in the domain and application engineering phase use of PLFaultCAT and the product-line SFTA technique. In particular, Figure 3 displays the commonalities, variabilities and dependencies associated with the wind speed detection, calculation and communication within the FWS. Figure 4 gives a portion of the Parameters of Variation document detailing the allowable options for the variabilities listed in Figure 3.

A product-line SFMEA searches the failure modes possible in the product line, determines their potential effects in the product line and establishes their potential effects on each member (Lutz et al., 1998). An excerpt of the SFMEA for the FWS is given in Figure 5. This portion of the SFMEA includes only those failure modes relevant to the requesting vessel receiving a corrupt wind speed report format from the FWS. Note that while this particular SFMEA concentrates mainly on the software failures of the FWS, it does include those hardware failures (which will typically contribute as leaf nodes) that contribute to the propagation of software failures. If a SFMEA exists, this analysis can produce the necessary domain knowledge to begin construction of the product-line SFTA using the prescribed steps detailed in Section 4.1. If a SFMEA does not exist, construction of the product-line SFTA proceeds directly to Step 2 below after assembling an intermediate node tree without the aid of a SFMEA.

Item	Failure Mode	Cause of Failure	Possible Effects
Reported Wind Speed Value	Inaccurate value	a. Wrong wind speed value outputted by software b. Deadline was missed by the FWS to report the current wind speed c. The wind speed report was reported in an incorrect format d. The wind speed reported is not trustable	A late, incorrect, unrecognizable or not trustable reported wind speed may create a hazard wished to be mitigated.
Unit Conversion Software	Incorrect value	a. Error in knots to KPH calculation b. Error in knots to MPH calculation	Inaccurate calculation for the wind speed algorithm.
Wind Speed Calculation Software	Incorrect value	a. Error in wind speed unit conversion software b. Corrupt input data to the algorithm c. Error in the wind speed calculation logic	An incorrect wind speed value may be outputted from the wind speed calculation module.
Wind Speed Output Value	Incorrect value	a. General software error b. Computer hardware malfunction c. Incorrect wind speed calculation	An incorrect wind speed value may be reported to the receiver.
Wind Speed Report	Incorrect value	a. Report is not in RAINFORM format b. Report is corrupted c. Wrong value units used in report	Can lead to an inaccurate and potentially unusable wind speed report by the receiver issued by the FWS.
	Incorrect format	a. A non-RAINFORM format was used to transmit the weather information from the FWS	The receiver will not recognize the reports format and will be unable to disseminate the data.
	Incorrect units	a. KPH was used for the wind speed rather than MPH or knots b. MPH was used for the wind speed rather than KPH or knots	The report may be in a format, particularly the units of the wind speed value(s), that the receiver will not recognize or will incorrectly use.
	Corruption	a. Radio transmission failure b. Communication transmission interference c. FWS computer memory corruption	The report may be corrupted when received by the receiver and may not be usable or used incorrectly.
Radio Transmitter	Malfunction	a. Generic radio transmitter malfunction	The report may be corrupted by a malfunctioning radio transmitter during the communication of the current weather report to a vessel or it may lead to a missed deadline in reporting the current weather (including wind speed).
Report Transmission	Interference	a. Unknown transmission communication interference	Interference incurred during the communication and transmission of the FWS weather report may corrupt the data being sent and may be unusable or inaccurate when received by the requesting vessel.
Computer Memory	Corruption	a. A computer memory failure occurred before the BIT check was performed to catch the failure.	Corrupted computer memory could possibly corrupt the data in the memory and corrupt the report the FWS is to send to any requesting vessel or it may cause a delay in the calculation algorithm.

Figure 5. A portion of the FWS Software Fault Modes and Effects Analysis

4.1 Product-Line SFTA Construction

The construction of the product-line SFTA using PLFaultCAT proceeds through three basic steps:

Step 1. Determine the root node and generate the intermediate node tree. As explained above, the root node hazard of any SFTA often derives from a preexisting hazards list or a list generated during the PHA phase (possibly from an SFMEA).

Causal events can be viewed as contributing events to the root node and are derived from the SFMEA or equivalent domain expertise. The SFMEA provides the causal events in the "Cause of Failure" column as well as the potential contributing nodes leading to the causal event. Gathering the causal events, we construct an intermediate node tree to establish the cause-event hierarchy. The intermediate node tree, while not necessary in the construction of a product-line SFTA, aids in jump-starting the organization and analysis of the product-line SFTA. Essentially, the intermediate node tree represents a typical fault tree without the Boolean logic gate relationships between causal events and effects. To determine the intermediate node tree using this process, we use the following PL-SFTA_CREATE algorithm starting with the root node event as the initial *event*:

```
PL-SFTA_CREATE(event):
    STEP 1 Create node in tree for event
    STEP 2 If node is not root node then
        STEP 2.1 Attach node to parent node
    STEP 3 Scan SFMEA "Possible Effects" column for event
    STEP 4 For each row with event found do
        STEP 4.1 event = event listed in "Cause of Failure" column
        STEP 4.2 PL-SFTA_CREATE   (event)
```

Following this algorithm, an intermediate node tree is created. Note that this intermediate node tree does not contain any Boolean logic gates, nor does it include any information typing the variabilities to the hazard. Applying this algorithm for the root node "Inaccurate wind speed reported" yields the tree depicted in Figure 6 as one of the four second-

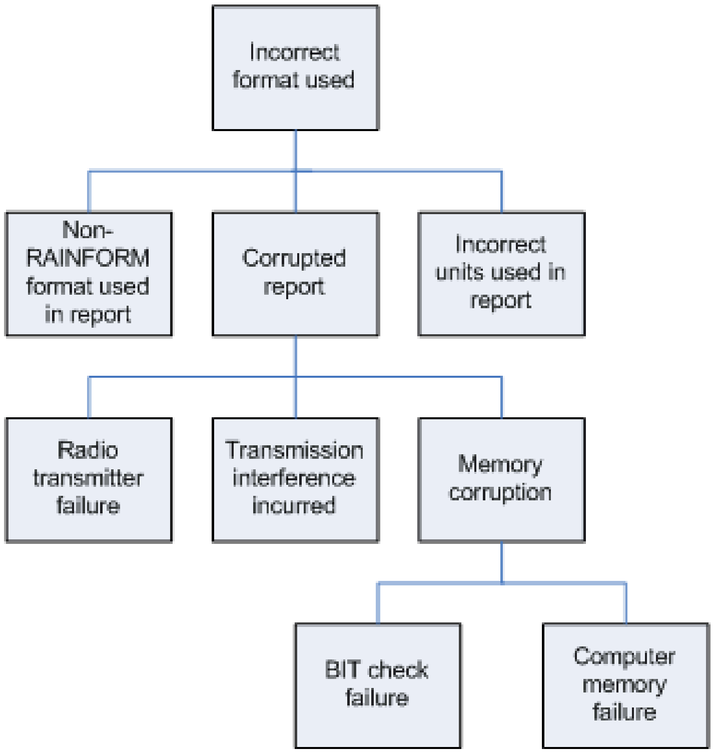

Incorrect format used
Non-RAINFORM format used in report
Corrupted report
Incorrect units used in report
Radio transmitter failure
Transmission interference incurred
Memory corruption
BIT check failure
Computer memory failure

Figure 6. FWS intermediate node tree

level subtrees that could potentially cause the root node hazard. Note that an "Inaccurate wind speed reported" event is not necessarily a hazard but is a failure, that might contribute to a hazard in some scenarios (e.g., if a ship or aircraft is dispatched in unsafe weather conditions based on this failure).

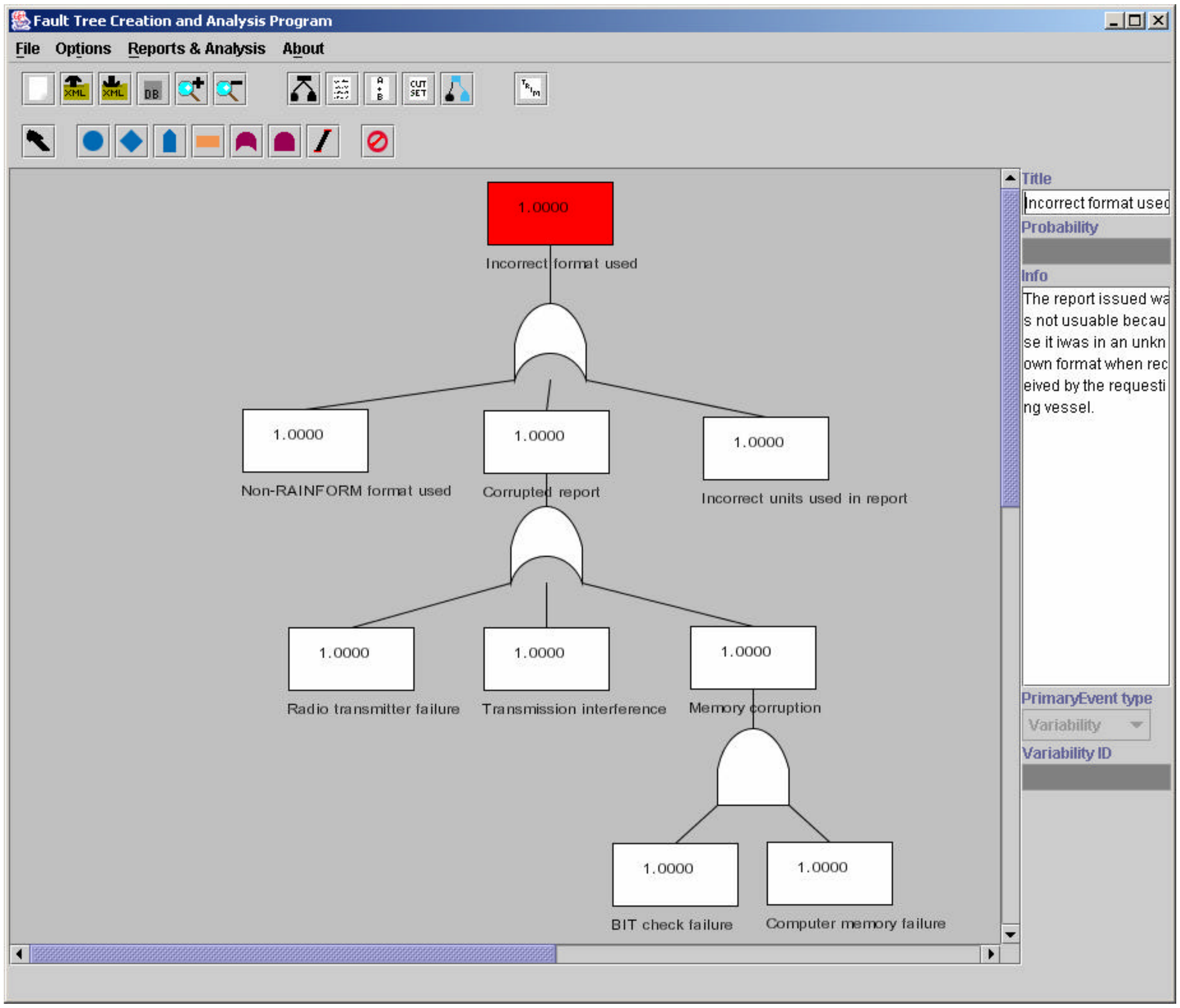
PLFaultCAT offers no distinct functionality to aid in completion of this step of the product-line software fault tree creation. In fact, PLFaultCAT cannot graphically construct a tree as shown in Figure 6 without Boolean logic gates relating causal events to the affected events (this is a result from inheriting the software architecture and functionality of the original FaultCAT tool). Rather, the intermediate node tree, constructed manually, acts as an input to PLFaultCAT.

Step 2. Refine the intermediate node tree and input into PLFaultCAT. The intermediate node tree produced in Step 1 can contain nodes that do not reflect the level of detail needed. A single node could actually be the effect of a combination of causes not captured in the SFMEA since SFMEA's typically cannot capture a series of causes leading to a failure event. Thus, domain expertise is needed to analyze the tree for completeness, capture additional events leading to a failure, and to refine nodes as needed. Using our intermediate node tree shown in Figure 6 for example, it may be desirable to further detail the causes of the node "Computer memory failure" or, if possible, reference a separate fault tree for this failure that details the causal factors.

Depending on the level of detail presented in the SFMEA, it may provide insight into what kind of logic gate should be applied to join children event nodes to their parents. Traditionally, SFMEA only considers a single failure at a time, thus implying logical OR gates throughout a SFTA. This is even more evident when the SFMEA distinguishes the variabilities from each individual failed Item/Event. However, our experience has shown that some detailed SFMEAs provided enough causal information to warrant a logical AND gate. For example, using our SFMEA, shown in Figure 5, as well as the intermediate node tree, shown in Figure 6, we can infer that the nodes "BIT check failure" and "Computer memory failure" must be joined by a logical AND gate in order to cause the "Memory corruption" node. Intuitively, this makes sense. Because of the advanced error trappings inherent in the FWS, the software will only incur corrupt memory if there indeed has been a memory failure *and* the FWS BIT check test, which is executed periodically, has failed to catch the corrupt memory. The caveat is that the SFMEA should only be used as a heuristic aided by domain knowledge and experts to produce the ultimate logic gate represented in the product line SFTA. Thus, the SFMEA should be mined to extract as much relevant information as possible to assist the construction of the product-line SFTA.

In addition to refining each node, we apply domain knowledge to determine the necessary logical combination of the children nodes to cause the parent node. This is a similar process to traditional fault tree analysis. Using the PLFaultCAT tool and applying Step 2 to the intermediate node tree, found in Figure 6, yields the intermediate software fault tree depicted in Figure 7.

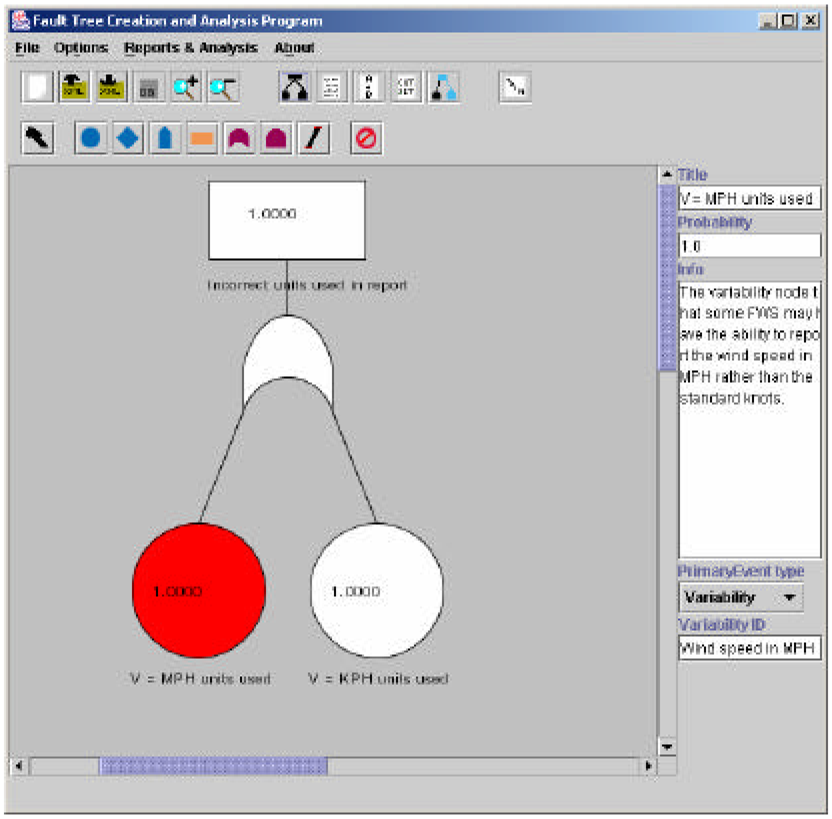
Aside from allowing the user to graphically construct a fault tree, PLFaultCAT allows an annotated description of each node so that the user can attach further details. This is especially advantageous in that it provides traceability to the hazard analysis. It also can be used to cross-check the completeness of the SFMEA by ensuring that all hazard events in the SFTA map to a cause or effect in the SFMEA (i.e., one-way traceability).



Title
Incorrect format used
Probability
Info
The report issued wa
s not usuable becau
se it iwas in an unkn
own format when rec
eived by the requesti
ng vessel.
PrimaryEvent type
Variability
Variability ID

Figure 7. FWS product-line intermediate software fault tree in PLFaultCAT

Step 3. Consider the influence of variabilities on all leaf nodes and tag each node accordingly. This is the crux of the product-line construction. In this step we employ a bottom-up approach to analyze each leaf node and determine which commonalities and/or variabilities contribute to causing the root node event to occur. In doing this, we associate the range of commonality and variability choices for any individual product-line member with how it might influence a particular hazard. Not every commonality or variability will have an influence or appear within any given fault tree. However, every leaf event node should have either an associated commonality, variability, and/or basic (primary) event.



Title
V= MPH units used
Probability
1.0
Info
The variability node t
hat some FWS may h
ave the ability to repo
rt the wind speed in
MPH rather than the
standard knots.
PrimaryEvent type
Variability
Variability ID
Wind speed in MPH

Figure 8. Depicting the influence of variabilities on an event in PLFaultCAT

When considering a variability's influence on a particular leaf node, we consider the parameters of variation allowed. While many variabilities are features that are simply present or not present in a product, some variabilities represent an allowable numerical or enumerated range for a particular feature. Considering the influence of a present or absent variability on an event is straightforward; we analyze the influence of the variability being present within the product and not functioning as designed.

If, however, we need to consider an enumerated or range type of variability, we must consider the various possibilities within the variability and their influence on fault tree events. For large ranges, safety analysis on each potential variability choice would be infeasible. Thus, class ranges are used to determine how different ranges could affect contributing events (Sommerville, 2004). Looking at the node "Incorrect units used in report" in our example, shown in Figure 7, and consulting the CA, shown in Figure 3, we conclude that this failure node can only occur if the FWS has the feature (variability) that it can report the wind speed in units other than knots. Thus, we annotate this node accordingly to indicate that the node "Incorrect units used in report" can only occur when either one of the variabilities (MPH units used or KPH units used) is present in a product line member. The representation of this is shown in Figure 8.

Using PLFaultCAT makes associating a commonality and/or variability with a failure node straightforward. The PLFaultCAT interface allows you to label the "Basic Event" nodes, depicted as circles, as a Variability (shown in Figure 8 under the heading "PrimaryEvent type") as well as defining a label or ID for the variability (the textbox under the heading "Variability ID"). In the example, in Figure 8, the variability (feature) has the label "V = MPH units used". The "Variability ID" describes the variability (feature) so that it will be recognizable later when selecting the variabilities to include in a new product line member. For this example, we simply annotate "Wind speed in MPH" to indicate that a product line member may or may not have this variability (feature).

The consideration of numerical ranges or values is particularly important because often not all values of a variability will contribute to a failure. Applying equivalence class partitioning and boundary value analysis concentrates on the fringe numbers and other frequently error-prone ranges to improve coverage of possible vulnerabilities. Using our FWS example, we can see from the Parameters of Variation document, shown in Figure 4, that we indeed have a numerical range variability: the number of wind speed sensors may vary between 1 to 5 sensors. For our product-line SFTA for the hazard "Incorrect wind speed reported" we encounter the situation where the variability of multiple wind sensors can cause a failure node and the commonality of having one wind sensor will not. PLFaultCAT accommodates this case by specifying the variability, as shown in Figure 8, by labeling it a "Variability" PrimaryEvent Type and specifying in the "Variability ID" field a label indicating that multiple wind sensors must be present in the product line member to cause the parent failure node. This same approach would be utilized for any enumerated variability.

Applying this step to our FWS example using the CA, shown in Figure 3, as well as the SFMEA, shown in Figure 5, we constructed a ninety-two node product-line SFT using PLFaultCAT for the root node hazard "Inaccurate wind speed reported". Specifically, the product-line SFT included fifty-five failure nodes and thirty-seven variability/commonality nodes (of which thirteen nodes were for variabilities and twenty-four for commonalities). Additionally, the product-line SFTA for this particular hazard included seven different variability instantiations contributing to failures.

Throughout the development and construction of the product-line SFTA we associate commonalities and/or variabilities with each leaf node in the intermediate node tree developed in Step 2. This process may yield both a commonality and variability being associated with a single failure node. In this case, intuition may suggest disregarding consideration of the variability since the causal event will always be present due to the presence of the associated commonality node. However, the risk of failure posed by the commonality may be mitigated while the risk posed by the variability remains. Hence, the variability must be retained to aid in the analysis of the product line, especially as the product line evolves.

Neither the construction of a product-line SFTA nor PLFaultCAT captures product-line dependencies. This is because the role of the product-line SFTA is to give as complete an account as possible of potential contributing causes to the root node. Note that the product-line SFTA does not enforce existing product-line dependencies. Instead, it represents all possible permutations of choices of values of product-line members and relies on dependency enforcement prior to the application engineering phase as in (Padmanabhan and Lutz, 2002).

Since SFTA adopts a slightly different perspective when viewing the product line, it is not uncommon to discover missing requirements. The construction of the product-line SFTA in PLFaultCAT may have some feedback effect on the CA in terms of discovering previously unidentified dependencies. Similarly, missing commonalities and variabilities, or incorrect parameters of variation may sometimes be identified via this process.

It is interesting to note that the influence of variabilities on hazards will not necessarily "sink to the bottom" of the fault tree but can instead be dispersed throughout the tree. Variabilities are commonly thought of as refinements of commonalities so the expectation is that they will only influence the root node from the lowest levels of the fault tree (Lu and Lutz, 2002). However, we found that this was not always the case. Variabilities, especially in software, are sometimes add-on features to the system rather than refinements of a commonality. Feature-oriented variabilities can spawn refinement variabilities of their own. Situations like this can lead to a product-line SFTA where variabilities are spread throughout the levels of the tree rather than clustered at the bottom.

It is important to note that the method outlined in Steps 1-3 is an iterative process that is repeated for all hazards in the hazards list. This will produce a set of product-line software fault trees.

4.2. Deriving Additional Safety Requirements from the Product-Line SFTA

The product-line SFTA can aid in the discovery of latent safety requirements by identifying high-risk variabilities and common causes and by identifying new constraints. The product-line SFTA construction process produces a set of fault trees with the corresponding contributing commonalities and variabilities attached to the appropriate leaf nodes. Using this set of software fault trees, we can identify or even tabulate the most frequent variabilities that contribute to the root node hazards. If certain variabilities contribute to root node hazards, additional safety requirements and/or hazard analysis may be warranted to mitigate their contribution to hazard nodes.

Any high-level event node within a product-line SFTA that has two or more variabilities connected by an AND gate may warrant a new constraint. Introducing a new product-line constraint limiting the variability combinations in this situation can preclude occurrence of this event node and potentially rid the product-line SFTA from this hazard altogether. However, care must be taken in deriving new product-line dependencies so that the product line is not too limited. The hazard severity as well as the existence of alternative preventive measures must be weighed against the addition of product-line dependencies.

Figure 9 shows a generic example of the derivation of a new product-line constraint from a logical AND gate connecting two variabilities. This example shows that we can mitigate the "Causal Event" node by restricting a system in the product line from having both V_1 and V_2 features. If this is found to be an acceptable solution, the product-line SFTA then retains the "Causal Event" subtree for completeness, but the occurrence of the subtree becomes essentially impossible.

Imposing additional safety requirements in the domain engineering phase improves the product-line specifications and reduces rework in the application engineering phase. The safety requirements and/or product-line dependencies derived from the product-line SFTA can reduce the analysis needed and reduce time-to-market for new products.

PLFaultCAT offers limited ability to assist the process of deriving additional safety requirements within the domain engineering phase aside from offering a visualization of the software fault tree. PLFaultCAT retains several functions from FaultCAT that may be useful in

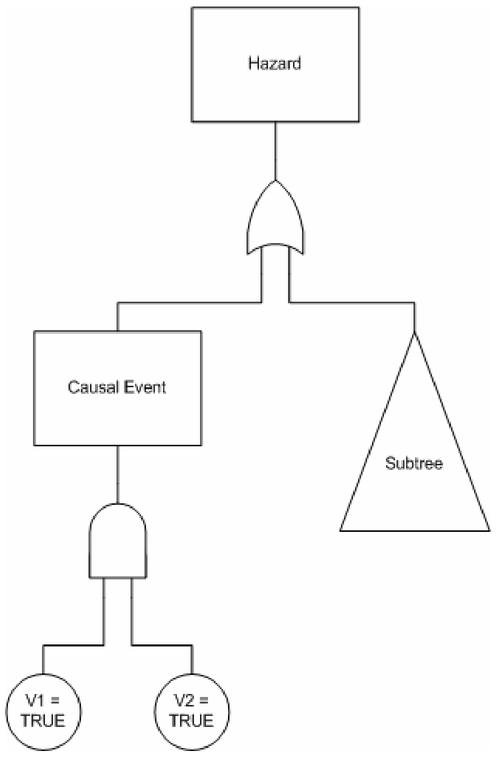


Figure 9. Generic product-line SFTA

the analysis of the product-line SFTA for new safety requirements. Functionality to display minimum cut sets is possible as well as the ability to assign probabilities to each node and then show the highest probability path(s) of the fault tree.

5. Application Engineering - Reusing the Product-Line SFTA

This section summarizes the reuse of the product-line Software Fault Tree Analysis (SFTA) developed in Section 4 using PLFaultCAT when building new product-line members. The Application Engineering phase, as illustrated in Figure 1, defines a new product-line member within the context of the previously defined requirements, commonalities and variabilities, and prunes the product-line SFTA, aided by PLFaultCAT, so that the previously performed safety analysis can be reused. This section also discusses the flexibility of the product-line SFTA in supporting product-line evolution as well as limits on reuse.

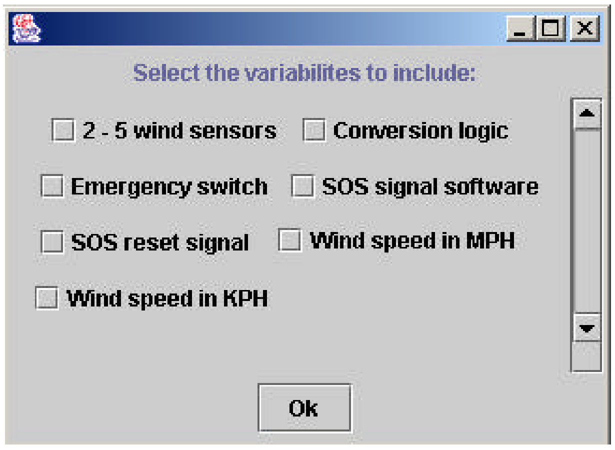


Figure 10. Selecting the product-line member's variabilities in PLFaultCAT

5.1. Pruning the Product-Line SFTA

In product-line SFTA we use a pruning process followed by a structured inquiry to develop a new product-line member's SFTA from the product line SFTA. Figure 6 presents a software fault tree for our Floating Weather Station (FWS) product family for one root hazard. The reuse of the product-line SFTA performed using PLFaultCAT for a new system in the product line has three basic steps: selecting the variabilities for a new product line member, deriving the product-line member SFTA, and applying domain knowledge, each of which are described below.

Step 1. Select the variabilities for new product-line member. Producing a product-line member entails a selection of which variabilities or features to include. This process can include an ordering of variability selection (e.g., according to domain model techniques in (Weiss and Lai, 1999)) or can leave the selection process to the system engineers. PLFaultCAT facilitates the selection of product-line member's variabilities through a checkbox window that presents all possible variabilities. Figure 10 shows an example of the variability selection window that will appear after a user clicks on the "Trim" button, found in the main toolbar, and selects the appropriate, already stored, product-line SFT xml file.

Typically, the selection of a set of variabilities does not guarantee a legal product-line member. Rather, the choice of variabilities must satisfy the previously established product-line dependencies and constraints. PLFaultCAT does not enforce nor check the dependencies prescribed in the Commonality and Variability Analysis (CA). Instead, other tools are capable of enforcing the dependencies and constraints detailed in the CA for large, complex product lines (Padmanabhan and Lutz, 2002). PLFaultCAT is used after the choice of variabilities has been determined to be legal.

For illustration purposes, we consider two different product-line members. The first consists of the features of having multiple wind sensors (between 2 and 5) and reporting the wind speed in KPH. The second product-line member contains an emergency switch feature. Thus, the features of these two products are as follows:

FWS A

- 2 - 5 wind sensors
- Conversion logic
- Wind speed in KPH

FWS B

- Emergency switch
- SOS signal software
- SOS reset signal

Both products also include all the commonalities described in the CA in addition to these features and satisfy the CA dependencies (see Figure 3).

Step 2. Derive the product-line member SFTA. After establishing and verifying a product-line member, we prune the product-line SFTA to create a baseline SFTA for the new system. The pruning process first uses a depth-first search to automatically remove the subtrees that have no impact on the product-line member being considered and then relies on a small amount of domain knowledge to further collapse and prune the SFTA. The algorithm starts with the root node as *node* and proceeds as follows:

```
PL-SFTA_SEARCH (node):
    STEP 1 If node is not a commonality leaf or a selected variability then
        STEP 1.1 Perform DFS for a selected variability or commonality node
        STEP 1.2 If DFS returns true
            STEP 1.2.1 For each child node do
                STEP 1.2.1.1 PL-SFTA_SEARCH(node)
        STEP 1.3 If search returns false then
                STEP 1.3.1 Remove subtree rooted at node
    STEP 2 Else if node is an unselected variability then
        STEP 2.1 Remove subtree rooted at node
```

A “selected variability” in our algorithm is an optional feature that is required in the new system. For example, a particular FWS may be required to calculate wind speed in KPH even though this feature is not a requirement of each system within the product line. An unselected variability, however, is an optional feature or a value of a variability not present in the new system.

PLFaultCAT implements this algorithm using the variabilities specified to include in the product-line member, as in Step 1. The tool processes the product-line SFT's XML file to create a new SFT including only those nodes associated with the commonalities and chosen variabilities for the new system. In Step 3 of the domain engineering phase, a label was attached to every variability by giving a variability name in the "Variability ID" textbox. It is this label, for the chosen variabilities, that is searched for in the XML file to decide whether a variability node should be retained. Upon completion of the PL-SFTA_SEARCH logic implemented in PLFaultCAT, the product-line member's SFT is displayed in the main PLFaultCAT viewer and a NewProductLineMember.xml file is saved to the working directory of PLFaultCAT.

The subtree shown in Figure 11 illustrates how the pruning algorithm executes within PLFaultCAT to remove irrelevant subtrees. Using the PL-SFTA_SEARCH algorithm for FWS A, we see that the subtree under consideration contains neither a failure node associated with a commonality or with a selected variability. Thus, the PL-SFTA_SEARCH algorithm used in PLFaultCAT will remove this entire subtree since it can have no influence on any of the parent failure nodes of this subtree. If, however, we consider FWS B for the subtree illustrated in Figure 11, we see that the entire subtree should be retained since the selected variabilities all can have an influence on the failure nodes in each path of the subtree. PLFaultCAT uses this logic in a depth-first fashion over the entire product-line SFT to derive the product-line member's SFT based on the selected variabilities.

Using the 92-node product-line SFT constructed in Section 4 on the hazard "Inaccurate wind speed reported", PLFaultCAT was used to automatically derive the fault trees for our representative product-line members: FWS A and FWS B. The initial execution of PLFaultCAT reduced the number of failure nodes by approximately 16% for FWS A and 13% for FWS B in the resulting product-line member fault trees.

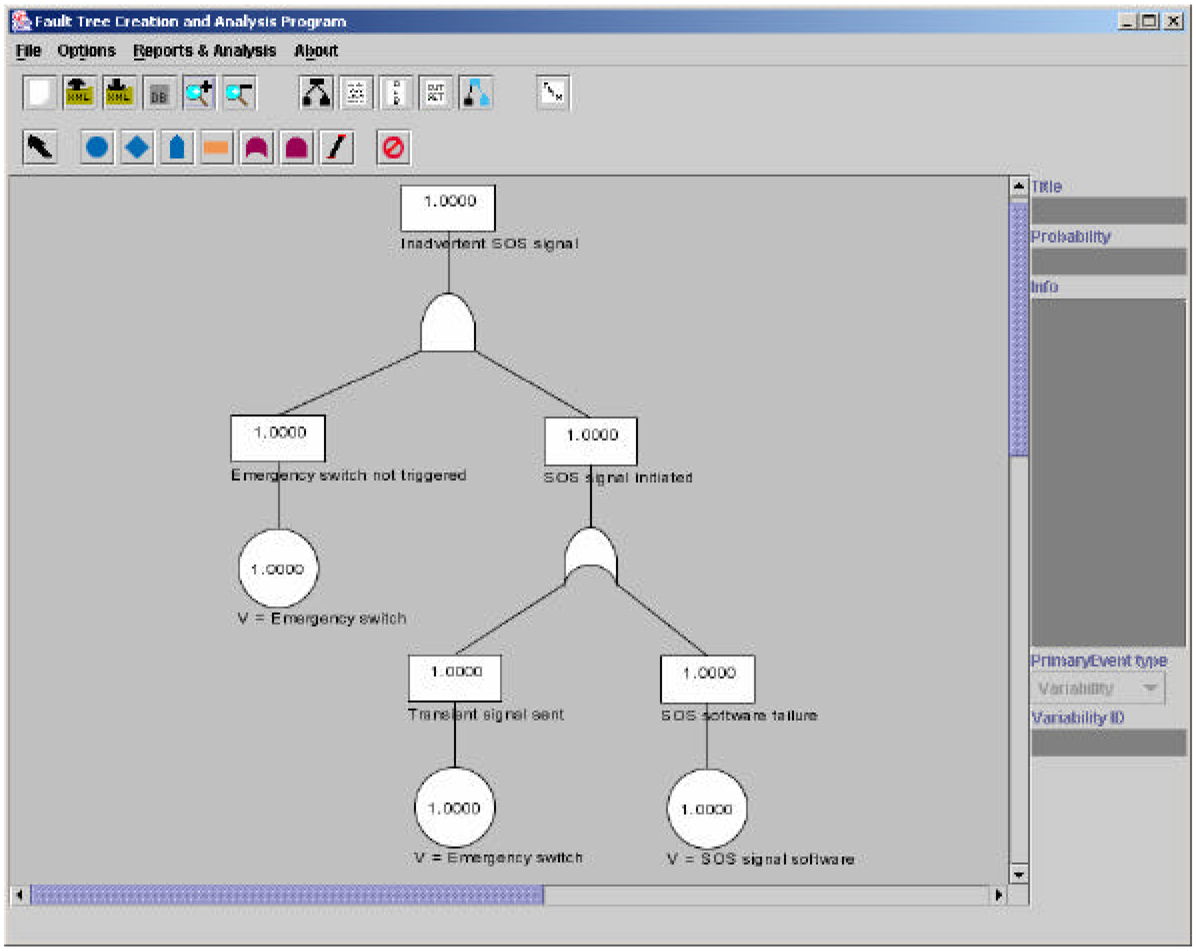


Figure 11. Prunning the product-line SFT in PLFaultCAT

The PL-SFTA_SEARCH algorithm errs on the side of caution since it only marks the subtrees that can be removed without review and does not actually do any pruning. This is advantageous from a safety perspective because the application of the algorithm simply indicates those subtrees where neither commonalities nor selected variabilities can be found in the subsequent children nodes. This algorithm then defers the actual pruning to the domain experts.

Step 3. Apply domain knowledge. After removing the subtrees that had no bearing on the product-line member under consideration, the tree may be able to be further pruned and/or collapsed within PLFaultCAT. However, this step requires domain knowledge. This also illustrates the limit to completely automated product-line SFTA reuse. Removal of subtrees/nodes will often lead to orphaned logic gates or other opportunities to safely simplify the product-line member's SFT, as shown in Figure 12.

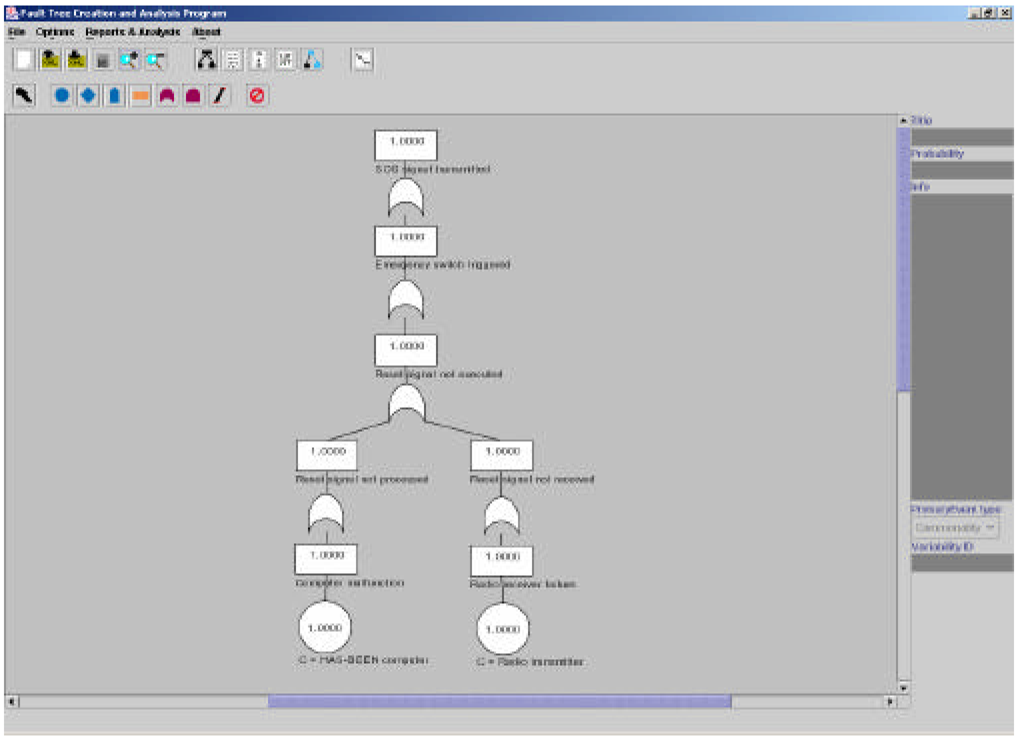


Figure 12. Resulting pruned product-line member SFT in PLFaultCAT

Collapsing orphaned OR gates is trivial. If there is only one causal event remaining, we collapse the lower event into the parent event. If there is only one commonality or variability leaf node remaining, we attach it to the parent event and remove the OR gate.

When AND gates are involved, we need to be more cautious. Intuitively, if at least one input line to an AND gate is removed, the output event is impossible. However, we found that this is not always the case and thus each removal of an AND gate warrants further scrutiny. For example, in another FWS fault tree an "Incorrect Wind Speed Reported" node occurs when both the "Deadline Missed" node AND the "Consensus Not Reached" node occurs. That is, even if a consensus is reached and reported, it is considered incorrect unless the deadline is made. Similarly, if the deadline is met but the consensus not reached, the report is incorrect. In this case, a product-line member that has no deadline (the existence of a deadline is a Boolean variability) must retain the "Consensus Not Reached" subtree. That is, even though the "Deadline Missed" branch is connected via an AND gate and the other branch can be pruned, we cannot automatically prune all subtrees under the AND gate.

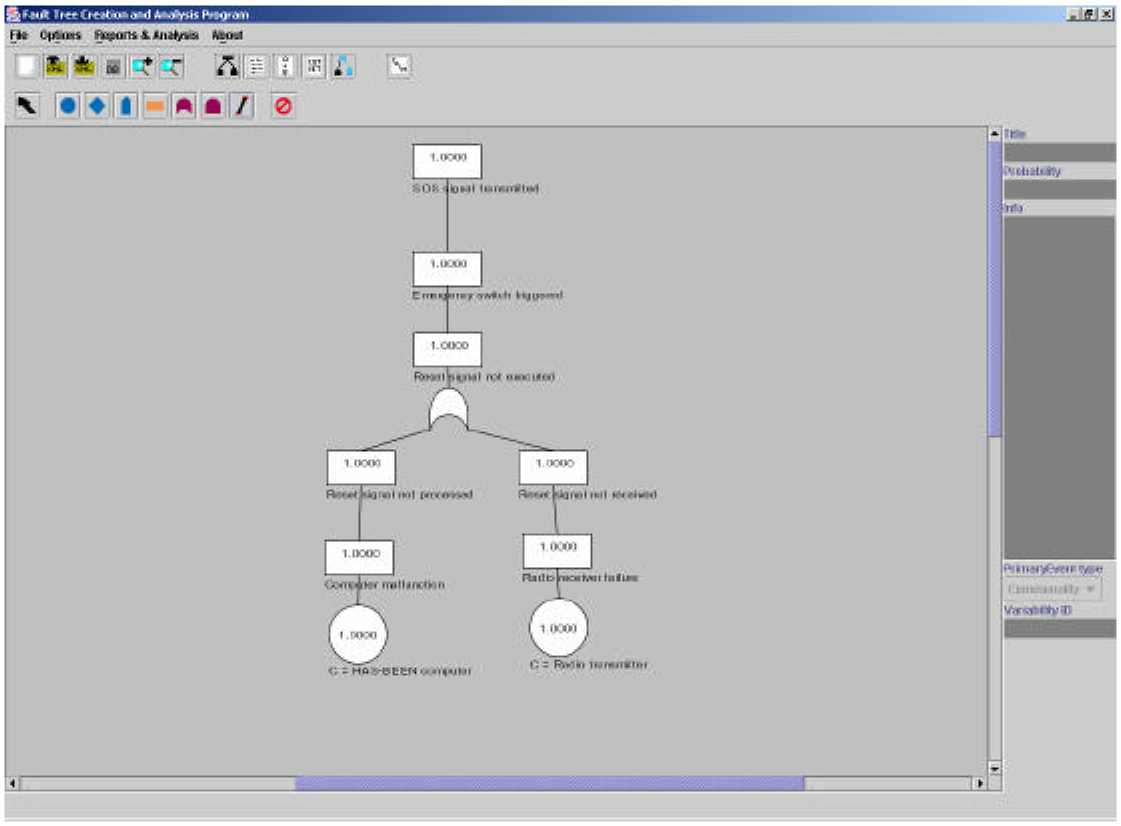


Figure 13. Cleaned product-line member SFT in PLFaultCAT

The clean up of the product-line member's SFT presented in this step is a manual process and must be pursued with utmost care. Enough information should be retained within the product-line member's fault tree to provide ample information for future hazard analysis and mitigation strategies. It is in this light that the subtree shown in Figure 12 reduces to the subtree shown in Figure 13 by removing the useless logic gates and connecting the failure nodes.

Looking back at the two representative product-line members, the clean-up process removed an additional five nodes from FWS A and two nodes from FWS B. Thus, the number of nodes in the SFT for the FWS A product-line member was reduced by over 25% from the number of failure nodes from the original product-line SFT with PLFaultCAT accomplishing most of the work automatically. Likewise, the FWS B product-line member was reduced by 17% from the original product-line SFT's failure nodes with PLFaultCAT doing a majority of the work.

The application of domain knowledge to the fault tree resulting from Step 2 is a beneficial step in the derivation of a product-line member's SFT because it removes the extraneous nodes and focuses attention on nodes that can potentially contribute to failures in a specific product-line member.

5.2. Product-Line Evolution

It is often the case that additional variabilities are added as features to the initial product line (e.g., as the consumer market changes). To safely include the new variabilities, we must perform a limited amount of domain engineering and hazard analysis to incorporate the new variabilities. In particular, new variabilities as well as new values for existing variabilities must iterate the relevant steps in the two-phase framework illustrated in Figure 1. This includes modifications to the requirements specification (as needed), as well as to the Commonality Analysis (CA) and SFMEA if they are affected.

In addition, the product-line SFTA is updated to incorporate the changes. If an SFMEA was constructed, the addition of variabilities can add new rows to the SFMEA or change the failures or effects in already existing rows in the SFMEA table. The PL-SFTA_CREATE algorithm, as detailed in Section 4.1, analyzes the new SFMEA rows and any additions to the preexisting SFMEA rows that can be influenced by the inclusion of the new variabilities. Following this process incorporates the new variabilities into the product-line SFTA by including their causal event nodes into the fault trees. The graphical view of the fault tree that PLFaultCAT provides makes updating the product-line SFT to incorporate new variabilities (features) and to derive a new product-line member's SFTA efficient enough for it to be practical for projects to maintain the fault tree as a current product-line artifact.

6. Evaluation and Analysis

In the domain engineering phase PLFaultCAT did not provide any significant advantages over other fault tree representation tools beyond providing the analyst with an additional opportunity to embed textual hazard analysis information into the fault tree. This allows a cross-check of the information provided in the fault tree with previously derived safety requirements, the Software Failure Modes and Effects Analysis (SFMEA) and other hazard analysis documents.

In the application engineering phase, however, PLFaultCAT provided significant advantages from a reuse perspective by exercising the pruning method outlined in Sections 5. For FWS A, thirteen failure nodes (24% of the total failure nodes) could be safely removed from the product-line SFT without losing necessary information. Of these, PLFaultCAT automatically removed eight of the thirteen, or 62%, using its PL-SFTA_SEARCH algorithm. The additional five of the thirteen required manual intervention using domain knowledge, as described in Step 3. For FWS B, nine failure nodes (16% of the total failure nodes) could be safely removed, seven of the nine automatically by PLFaultCAT (77%) and two additional nodes manually. These metrics reflect the effort saved in reuse of the SFT.

A concern for performing safety analysis on safety-critical product lines is whether the technique is scalable as the product line grows more complex by incorporating more variabilities and product-line members. While the work reported here only considered a small product line, it appears that our method and tool will scale adequately as the product line grows more complex. This is because most of the added complexity in a large product line lies in the domain engineering phase when the product-line SFTA is constructed. Since, the construction of the product-line SFTA described in Section 4 relies heavily on the aid of a product-line SFMEA. It appears that the scalability is at least as robust as that of the SFMEA. Additionally, it should be clear that the reuse of the product-line fault tree approach is far more efficient especially for large product lines than to serially construct SFTAs for each of the desired product-line members of a product line.

The communicability of a product-line SFTA created in PLFaultCAT with other applications is high since PLFaultCAT provides a user with three different views of any given fault tree: a standard graphical fault tree view, an XML file view and a text-based view. This variety of product-line SFTA views should allow PLFaultCAT's integration into other safety analysis techniques and tools. The XML output file utilized in PLFaultCAT supports straightforward linking with existing static analysis tools. For example, the use of a product-line SFT created in PLFaultCAT with other applications (such as Relex or DECIMAL) would at most only require a translation program to mediate the format of the XML file.

PLFaultCAT should be viewed mainly as an application engineering asset. Thus, within the domain engineering phase of a typical safety-critical product line, PLFaultCAT only provides a means to graphically aid the analyst in representation of the product-line fault tree. Where PLFaultCAT falls short is in the ability to enforce a product line's dependencies as detailed in a Commonality and Variability Analysis (CA) and in the ability to provide any ordering mechanism in the way the variabilities are selected for any given product-line member. While these processes are vital to the entire product line engineering process, PLFaultCAT cedes this functionality to other available tools such as DECIMAL (Padmanabhan and Lutz, 2002).

7. Conclusion

This paper described an extension of the traditional Software Fault Tree Analysis (SFTA) technique to an entire product line with the support of a software tool, PLFaultCAT. This extension supports construction of a product-line SFTA in PLFaultCAT from common hazard analysis assets during the domain engineering phase. The paper described how new safety requirements can be discovered through the introduction of product-line constraints. The paper also presented the pruning technique developed and implemented in PLFaultCAT during the application engineering phase to derive the SFTA for single product members of the product line. Planned future work will investigate the approach's reuse value and scalability through a large, industrial case study of a safety-critical medical product line. It is hoped that the use of this technique and tool will help improve the safety analysis of critical product lines.

8. Acknowledgements

This research was supported by the National Science Foundation under grants 0204139 and 0205588.

9. References

Ardis, M.A. and Weiss, D.M. 1997. Defining Families: The Commonality Analysis. *Proc. 19th Int'l Conf. Software Engineering* (ICSE '97), Boston, MA, pp. 649-650.

Burgess, M. 2003. Fault Tree Creation and Analysis Tool: User Manual. http://www.iu.hio.no/FaultCat (current, May 2004).

Clements, P. 2002. Being Proactive Pays Off. *IEEE Software*, 19(4):28, 30.

Clements, P. and Northrop, L. 2002. Software Product Lines. Boston: Addison-Wesley.

Coppit, D. and Sullivan, K.J. 2003. Sound Methods and Effective Tools for Engineering Modeling and Analysis. *Proc. 25th Int'l Conf. Software Engineering* (ICSE '03), Portland, OR, pp. 198-207.

Dehlinger, J. and Lutz, R.R. 2004. Software Fault Tree Analysis for Product Lines. *Proc. 8th IEEE Symposium on High Assurance Systems Engineering* (HASE '04), Tamp, FL, pp. 12-21.

Doerr, J. 2002. Requirements Engineering for Product Lines: Guidelines for Inspecting Domain Model Relationships. Diploma Thesis, University of Kaiserslautern.

Douglass, B.P. 1999. Doing Hard Time: Developing Real-Time Systems with UML Objects, Frameworks and Patterns. Boston: Addison-Wesley.

Hansen, K.M., Ravn, A.P. and Stavridou, V. 1998. From Safety Analysis to Software Requirements. *IEEE Trans. on Software Engineering*, 24(7):573-584.

Kang, K.C, Kim, S. Lee, J. Lee, K. 1999. Feature-Oriented Engineering of PBX Software for Adaptability and Reusability. *Software Practice and Experience*, 29(10):167-177.

Leveson, N.G.1995. Safeware: System Safety and Computers. Boston: Addison-Wesley.

Lu, D. and Lutz, R.R. 2002. Fault Contribution Trees for Product Families. *Proc. 13th Int'l Symp. Software Reliability Engineering* (ISSRE '02), Annapolis, MD, pp.231-242.

Lutz, R.R. 2000. Extending the Product Family Approach to Support Safe Reuse. *Journal of Systems and Software*, 53(3):207-217.

Lutz, R.R. 2000. Software Engineering for Safety: A Roadmap. *Proc. of the Conference on the Future of Software Engineering*, New York, NY, pp. 213-226.

Lutz, R.R., Helmer, G.G., Moseman, M.M., Statezni, D.E. and Tockey, S.R. 1998. Safety Analysis of Requirements for a Product Family. *Proc. 3rd Int'l Conf. on Requirements Engineering* (ICRE '98), Colorado Springs, CO, pp. 24-31.

Lutz, R.R. and Woodhouse, R.M. 1997. Requirements Analysis Using Forward and Backward Search. *Annals of Software Engineering*, 3:459-474.

Padmanabhan, P. and Lutz, R.R. 2002. DECIMAL: A Requirements Engineering Tool for Product Families. *Proc. 2002 Int'l Symp. Software Reliability Engineering for Product Lines* (REPL '02), Essen, Germany, pp. 45-50.

Pai, G.J. and Dugan, J.B. 2002. Automatic Synthesis of Dynamic Fault Trees from UML System Models. *Proc. 13th Int'l Symp. Software Reliability Engineering* (ISSRE '02), Annapolis, MD, pp. 243-254.

Schmid, K. and Verlage, M. 2002. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software*, 19(4):50-57.

Sommerville, I. 2004. Software Engineering. Boston: Pearson Addison-Wesley.

Weiss, D.M. and Lai, C.T.R. 1999. Software Product Line Engineering: A Family-Based Software Development Process. Boston: Addison-Wesley.