Bi-Directional Safety Analysis of Product Lines

Qian Feng
Department of Computer Science
Iowa State University
qianfeng@cs.iastate.edu

Robyn R. Lutz
Department of Computer Science
Iowa State University and
Jet Propulsion Laboratory
rlutz@cs.iastate.edu

Abstract

As product-line engineering becomes more widespread, more safety-critical software product lines are being built. This paper describes a structured method for performing safety analysis on a software product line, building on standard product-line assets: product-line requirements, architecture, and scenarios. The safety-analysis method is bi-directional in that it combines a forward analysis (from failure modes to effects) with a backward analysis (from hazards to contributing causes). Safety-analysis results are converted to XML files to allow automated consistency-checking between the forward and backward analysis results and to support reuse of the safety-analysis results throughout the product line. The paper demonstrates and evaluates the method on a safety-critical product line subsystem, the Door Control System. Results show that the bi-directional safety analysis method found both missing and incorrect software safety requirements. Some of the new safety requirements affected all the systems in the product line while others affected only some of the systems in the product line. The results demonstrate that the proposed method can handle the challenges to safety analysis posed by variations within a product line.

Keywords: Product lines; Software safety; Software architecture; XML; Reuse.

1. Introduction

As product-line engineering becomes more common, more safety-critical product lines are being built. A product line is a set of systems developed from a common set of core requirements and sharing a suite of common traits among the members [Ardis and Weiss, 1997; Weiss and Lai, 1999]. Examples of safety-critical product lines include embedded medical devices such as pacemakers, space telescopes, power-plant control systems, and some industrial robots.

The potential for reuse among the systems in a software product line extends beyond code reuse. Reuse of software assets currently includes product-line requirements specifications, product-line core architecture, product-line test suites and product-line performance analyses.

This paper describes results from an investigation into how, and to what extent, product-line safety analyses can be performed and reused as a product-line asset. That is, we are interested in the potential for reuse of the safety analysis among the members of a safety-critical product line. The motivation for this research is to improve the safety-analysis techniques available to developers of commercial, safety-critical product lines.

It is important to note that safety is a property of a single system, not of a set of systems. Thus, any safety analysis done during the early domain engineering of the product line (i.e., when the entire product line is being defined) must be re-evaluated, adjusted, and completed during application engineering (i.e.,

when each individual system is built). Some preliminary results regarding the reuse of safety analyses during application engineering have appeared in [Dehlinger and Lutz, 2004; Lu and Lutz, 2002]. In this paper we focus instead on the process of developing a product-line safety analysis for the domain engineering phase of safety-critical software product lines.

The paper extends the Bi-Directional Safety Analysis (BDSA) method [Lutz and Woodhouse, 1997] to product lines. The BDSA method combines a forward search from potential failure modes to their effects with a backward search from feasible hazards to the contributing causes of each hazard. The forward search is similar to a Software Failure Modes and Effects Analysis (SFMEA); the backward search is similar to a Software Fault Tree Analysis (SFTA). The combination of the forward and backward search has proven effective in discovering latent safety requirements.

The work described in this paper investigates two major challenges to extending the BDSA method to product lines: how to adequately understand and specify the safety consequences of the variations among the members of the product line, and how to structure the process such that the safety analysis is derived from, and traceable to, the product-line requirements and design.

In order to address these challenges in a way that is likely to be used by industry, the safety-analysis method presented in this paper is grounded in the standard artifacts of the product-line domain-engineering process. These domain-engineering assets are: (1) the Commonality and Variability Analysis that specifies both the requirements shared by all the systems in the product line and the variations among the systems' requirements; (2) the product-line architecture that forms the shared, core software architecture for all the systems and supports the required variations; and (3) the product-line use cases and scenarios that specify the range of uses and the sequences of events that some or all of the systems in the product line may experience.

Grounding the safety analysis in the domain-engineering products has several benefits. First, it supports documented traceability from the extended commonality analysis to the safety analysis and is requisite for future automated updating of the safety analysis as the product line evolves. Second, linking the safety analysis to the products that capture the subtleties of the domain provides more complete handling of variations, the rationales for the variations and the consequences of the variations in the safety analysis. Third, using standard domain-engineering assets promotes readier adoption of the safety-analysis method by companies building safety-critical, software product lines and can lower the cost of performing enhanced safety analyses on these product lines. The first two benefits are demonstrated in the paper by application of the safety-analysis method to the Door Control System, a safety-critical subsystem of the Smart Home product line.

Figure 1 shows an overview of the analysis method developed in this paper with the Extended Commonality Analysis driving the bi-directional Safety Analysis in the lower half of the figure. Our method consists of seven steps:

Step 1: Performed Commonality and Variability Analysis to specify the requirements for the given product line.

Step 2: Developed the architecture design and sequence diagrams from the product-line requirements.

Step 3: Extended the Commonality and Variability Analysis based on the results of the Commonality and Variability Analysis and the Architecture Design diagrams.

Step 4: Constructed the Software Fault Tree Analysis from the results of the Extended Commonality and Variability Analysis.

Step 5: Constructed the Software Failure Mode and Effect Analysis from the results of the Extended Commonality and Variability Analysis.

Step 6: Translated the results of the Software Fault Tree Analysis and Software Failure Mode and

Effect Analysis into XML files.

Step 7: Compared the resulting XML files using Xlinkit.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 describes the product line commonality and variability analysis. Each step in this and subsequent sections is illustrated with examples from the Door Control System. Section 4 shows how to extend the commonality analysis with information from the architectural design and the scenarios' sequence diagrams. Section 5 describes how the bi-directional software safety analysis uses the extended commonality analysis to guide and structure it. Section 6 evaluates the method, both by automated consistency checking of the forward vs. backward safety analysis results, and by discussing the missing or incomplete software safety requirements for the product line found by this method. Section 7 briefly summarizes the results and provides some concluding remarks.

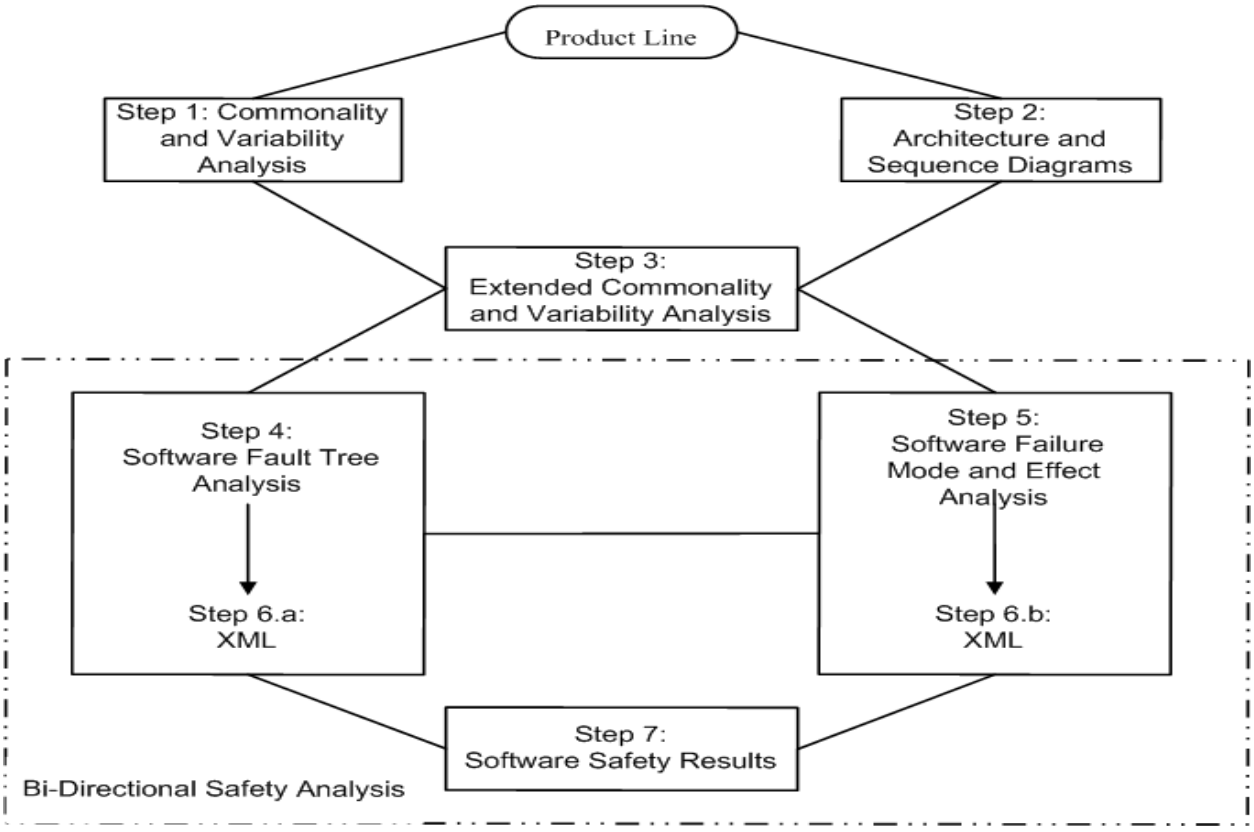

Figure. 1. An Overview of the Safety Analysis Method

2. Related work

The work described here pulls together related work in two main areas: product line engineering and software safety.

In the area of product line engineering, our work draws on recent work in product line requirements and in product line architectures. We follow Ardis and Weiss, and Weiss and Lai in using a Commonality and Variability Analysis to identify and specify product line requirements [Ardis and Weiss, 1997; Weiss and Lai, 1999]. Section 3 describes our use of their approach in some detail. Our bi-directional safety

analysis method is also compatible with alternative approaches to product line requirements analysis, such as PuLSE [Bayer et al., 1999] and FORM [Kang et al., 1998]. However, because these approaches can enforce an ordering on the choices of features when a new product line member is built, we prefer the partial ordering of the Commonality and Variability Analysis.

Product line architecture is an extensively studied field. Bosch discusses how to develop the architectures to a product line and how to revise the product line architecture for different products. After analyzing all the products, an architecture design and its components are developed in three steps. The first step is to develop a product line architecture supporting the functional and quality requirements of the product line; the second step is to revise the product line architecture design to specific designs for different products; and the third step is to evolve the architecture design for new requirements and new products [Bosch, 2000]. Bass says a product line architecture design is built on the three points of "identifying variation", "supporting variation", and "evaluating for product line suitability". Product line architecture support for variation is represented by "inclusion or omission of elements", "inclusion of a different number of replicated elements", and "selection of versions of elements that have the same interface but different behavioral or quality attribute characteristics" [Bass, 2003]. Egyed points out a product line architecture provides "generic information common" to all the products in the product line and includes "a certain amount of ambiguity" in order to support variations in the individual product. "An individual architecture design is an instantiation of the product line architecture, which is less ambiguous" [Egyed et al., 2000]. The architecture for a product line is "a generic architecture from which the individual product architectures can be derived" and it provides two fundamental usages: one is the architecture for a whole product line can "capture the important aspects of the product line"; the other is an individual product's architecture can be instantiated from the product line architecture [Perry, 1998]. A "core" architecture or "baseline" architecture can be derived from taking the essential features of the product line [Lutz, 2000; Lutz and Gannod, 2003]. An advantage of the "core" product line architecture is that new products can be added to the product line as long as they meet the basic design constraints.

The use of UML, use cases, scenarios, and sequence diagrams has been widely studied for product lines. Bayer, e.g., uses scenarios to determine architectural requirements [Bayer et al, 2000]. And Clauß extends UML to support feature diagrams and variability in the standard kinds of UML diagrams [Clauß, 2001]. The KobrA method is a component-based development for product lines and aims at increasing the reuse of the product line. It describes how to use UML diagram model components for product lines [Atkinson et al., 2002]. John and Muthig describe how use cases can be applied for modeling the requirements for a system family and how a particular single-system use case approach can be extended to capture product line information and especially variability [John and Muthig, 2002].

Besides product line engineering, the other area of related work for our study was software safety, software safety analysis, investigation how software can jeopardize or contribute to the safety of a system [Leveson, 1995]. To date, there has been relatively little work directed specifically at the challenges of safety-analyses for product line. Initial work by Dehlinger and Lutz [Dehlinger and Lutz, 2004], and by Lu and Lutz [Lu and Lutz, 2002] have shown how software fault tree analysis, a successful safety analysis technology for single system, can be extended to product lines and reused, with caveats, for a new member of the product line. Progress has been made in ensuring that quality attributes are preserved in product line [Bass, 2003; Bosch, 2000].

Our work differs from previous work in two ways: (1) the focus to date has been on architectural analysis rather than on requirements analysis, as we do here, and (2) prior work has not clearly distinguished safety from other quality attributes such as performance, modifiability, safety, reliability, availability, and testability. Our research emphasizes the safety analysis as a unique property that must be assured across the product line.

3. Commonality and Variability Analysis

In this section, we describe Step 1 of the safety-analysis process summarized in Figure 1. This step uses the product-line Commonality and Variability Analysis to specify the requirements for the product line of interest.

Safety analysis of a product line begins with analysis of the requirements. A product line is a set of products that share common aspects and differ from each other through some variabilities [Ardis and Weiss, 1997; Weiss amd Lai, 1999]. Through analysis of the requirements specification, we can identify software requirements related to safety and analyze them for completeness, consistency, and correctness.

The Commonality and Variability Analysis (CA) of a product line provides a requirements specification for the product line. A CA typically includes three parts: terminology, commonalties, and variabilities. The terminology is a "dictionary of standard terms"; the commonalities are "a list of assumptions that are true for all family members"; the variabilities "define how family members may vary" and "the scope of the family by predicting what decisions about family members are likely to change over the lifetime of the family" [Ardis and Weiss, 1997; Weiss amd Lai, 1999].

We demonstrate our methodology using the Door Control System for a Smart Home. The Door Control System is a safety-critical product line in that the software must function correctly to prevent intruders from entering and must respond correctly to life-threatening scenarios such as fires. A Smart Home system serves as an invisible housekeeper: it has sensors and agents to interact with humans and the environment to offer people convenience and safety. For example, the entrance doors can be opened only by inputting fingerprints or voiceprints [Cook, 2003; Youngblood, 2002; Smart-home project, Finland; Fellbaum and Hampicke]. We restrict our discussion in this paper to a Door Control System software product line with three products: a FrontDoor, a BedRoomDoor, and a SecurityDoor.

This section provides examples from the Commonality and Variability Analysis for the Door Control System product line derived from the detailed descriptions of these three products. The CA consists of the terminology used, the commonalities, the variabilities, and the dependencies among the variabilities. The dependencies are constraints that the choice of one features places on the choices of other features [Doerr, 2002]. Note that we here exclude any non-behavioral commonalities and variabilities to focus on the software. The CA serves as a requirement specification for the product line and as an input to the product line's architecture design.

3.1. Terminology

Table 1
Terms for Commonality and Variability Analysis

Name	**Explanation**
Door alarms	The alarm of the door which will be triggered by the illegal entry of the door.
Registration	A person's ID (fingerprints or voiceprints) is input to the database to be recognized later. (Note that by "fingerprint registration", we mean the input of fingerprint images into a database. Some researchers instead use "registration" to refer to the "alignment" of a fingerprint image with stored images.)
Recognition	The door tests the ID to see if this person has access permission. If so, the door will open for this person, otherwise the door will not open.
Family member	A person who has access permission for the FrontDoor

Correct people/person	Person(s) who should be let in upon requesting a door's opening.
Wrong people/person	Person(s) who should not be let in upon requesting a door's opening.
Fire alarm	The alarm that indicates the fire
Forced entry	Perceived force from the outside environment to try to break the door or the lock, e. g., through impact or pressure on the door's surface above a limit.
Illegal input	Wrong ID.
Lock inside	If the door is locked inside, nobody can open the door from outside, even if this person belongs to the set of correct people.
People pass	A person's whole body passes the door from one side of the door to the other side.

3.2. Commonalities

C1. Accept family members' registration
C2. Recognize correct people
C3. Open door for correct people from outside
C4. Open door for people from inside
C5. Close door after people pass
C6. Sound door alarm upon forced entry
C7. Respond to the fire alarm

3.3. Variabilities

V1. Methods of recognition: fingerprint or voiceprint. FrontDoor: fingerprint; BedRoomDoor: voiceprint; SecurityDoor: fingerprint and voiceprint.
V2. Methods of registration: fingerprint or voiceprint. FrontDoor: fingerprint; BedRoomDoor: voiceprint; SecurityDoor: fingerprint and voiceprint.
V3. Whether or not the door can be locked inside.
V4. Open/close doors when fire alarm on (when BedRoomDoor door and SecurityDoor are closed while the fire alarm is on, it will be open by pushing).
V5. Methods of opening doors inside (weight, oral command, or input IDs).
V6. Methods of triggering doors' alarm: the FrontDoor's alarm can be triggered by the wrong fingerprint input three times; the SecurityDoor's alarm can be triggered by wrong IDs input once. The BedRoomDoor's alarm will not be triggered by the wrong fingerprint inputs.

3.4. Dependencies among variabilities

(1) The method of recognition must be the same as the method of registration.
(2) The door's response to the fire alarm is dependent on the method of recognition.
(3) Whether the door can be locked inside is dependent on the method of recognition.
(4) The method of opening the door from inside is dependent on the method of recognition.
(5) The ways to trigger the door's alarm is dependent on the method of recognition.

4. Extended Commonality and Variability Analysis Using Architecture Design and Sequence Diagrams

In this section, we describe Step 2 and Step 3 of Figure 1. These steps develop the architecture design and sequence diagrams, and use these to extend and refine the Commonality and Variability Analysis.

In order to capture the domain knowledge needed to perform product line safety analysis, information about design choices and constraints must be conjoined with the requirements specification. In particular, insight into how the system can "go wrong" and into the rationales behind the design choices is needed. The components in a system and the communication among them can be identified by the architectural design. Since a product line architecture is structured for reuse, its specification displays interactions and data transformation in term of handling of potential variations among the product line members. These architectural details provide a structure for assembling and evaluating the Software Failure Modes Effects Analysis and Software Fault Tree Analysis used in the safety analysis in Section 5. Since many safety-related scenarios involve particular sequencing of events of interactions, the safety analysis also needs a dynamic view of the system's execution. This is achieved by connecting a sequence-diagram perspective to the Commonality and Variability Analysis.

In this section, we introduce a core architecture for the product line and derive individual architectures for product line members. We extend the Commonality and Variability Analysis developed in Section 3 with information from the architectural design and from the scenarios' sequence diagrams. The resulting Extended Commonality and Variability Analysis (XCA) provides the information needed to perform safety analysis on the product line. Furthermore, the XCA provides a foundation for reusing the safety analysis for new members of the product line in the future.

4.1. Architecture of DCS product line

The core architecture for a product line is a generic architecture, which not only captures the important common features of the product line, but also can be instantiated to be an individual product's architecture [Perry, 1998]. New products are added to the product line and reuse the architecture design, corresponding components, and corresponding safety analyses. Figure 2 shows a core architectural design for the Door Control System product line. The system is divided into two parts. One is the Central Control System; the other part includes the agents that communicate with the outside environment to detect changes and to provide required responses.

The Central Control System has five components: RegistrationComponent(Regis), RecognitionComponent(Recog), IllegalEntryComponent(IEC), DoorOpenCloseComponent(DOC), and FireAlarmController(FAC). The detectors sensing the outside environmental inputs are RegistrationDetector, RecognitionDetector, OpenFromInsideDetector, ForcedEntryDetector, PeoplePassDetector, and FireAlarmSensors. The responders that affect the environment are the Door's alarm and the Door (the door's position and the lock status). There is also a database connected to the central Control System that stores the ID data.

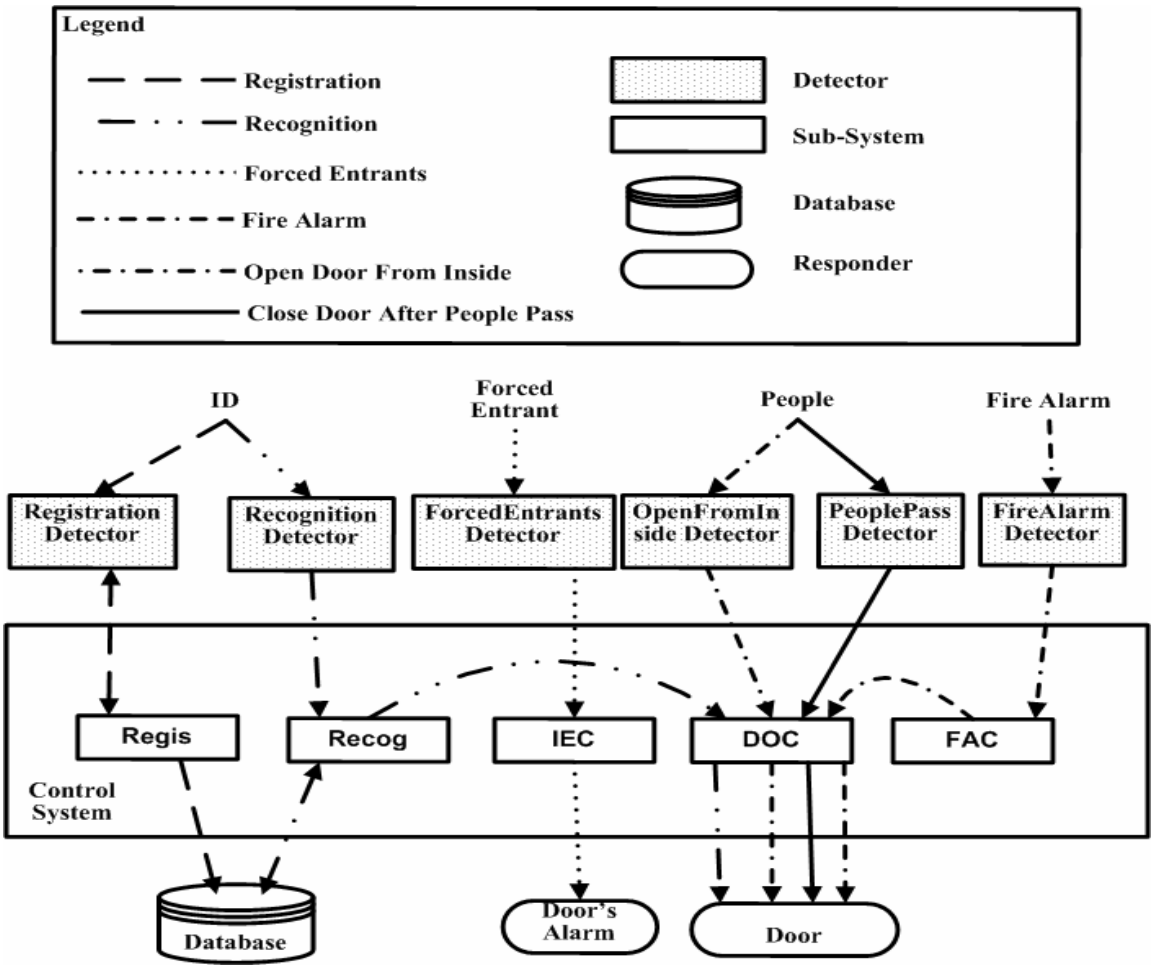

Figure. 2. The Core Architecture of DCS

Figure 3 is an individual architecture for the FrontDoor software system derived from the product line core architecture. The most obvious difference from the core architecture is that the sensors have been instantiated to some specified detectors. For example, a "RegistrationDetector" in the core architecture is instantiated to an "F_registration detector" (Fingerprint) in the FrontDoor, a "V_ recognition detector" (Voiceprint) in the BedRoomDoor, and an "FV_registration detector" (both fingerprint and voiceprint) in the SecurityDoor. An "F_registration detector" is a camera to catch fingerprints; a "V_ recognition detector" is a sound detector to catch voiceprints; and an "FV_registration detector" is a detector that catches fingerprints and voiceprints at the same time.

Another difference between the FrontDoor architecture and the core architecture is the connector in Figure 3 between the RecognitionComponent (Recog) and the IllegalEntryComponent (IEC), representing V6 of the variabilities in the Commonality Analysis. If there is an illegal ID input, the RecognitionComponent will send a signal to the IllegalEntryComponent, which will sound the door's alarm. Thus, the individual architecture for the FrontDoor requires an additional connector and some added functionality to generate and respond to the message that an illegal entry has been attempted.

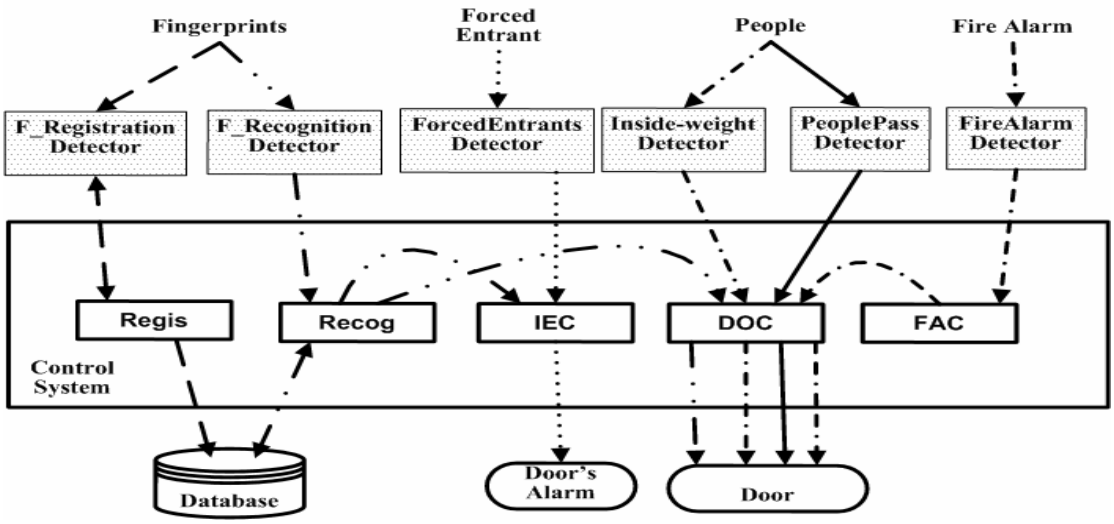

Figure. 3. Individual architecture for FrontDoor product

4.2. Scenarios

Identification of the data flow and event sequencing is very important to the quality of the safety analysis, since our SFTA and SFMEA are built based on the communication of the components in the architecture diagrams. Scenarios are sequences of system activities used to illustrate system behaviors and "operational instances of system uses" by showing the interactions between the objects, including the messages or events flowing between the components [Allenby and Kelly, 2001; Goseva-Popstojanova et al., 2003; Sommerville, 2000]. Through analysis of scenarios and construction of the sequence diagrams, we not only obtain a deeper understanding of the product line, its requirements specification, architecture, and the interactions between components, but also identify all data transferred between components and all events.

For example, for the Door Control System product line, we derived the following seven use cases by associating every commonality and every variability not included in the commonalities with a use case.

1. *Registration*: registering the users' ID to the Door Control system (from Commonality C1)
2. *Entry*: entering the house from the outside (Including recognition from outside, opening the door, closing the door after the people pass, also including the illegal entrant handling) (from Commonalities C2, C3, C5, and C6)
3. *Exit*: exit the house from the inside (Including recognition from inside, opening the door, closing the door after the people pass, also including the illegal going out handling) (from Commonalities C4, C5, and C6)
4. *Fire alarm*: the door's response to the fire alarm (from Commonality C7)
5. *Bolt*: lock door from inside (from Variability V3)

Each use case has one main scenario and other miss-use scenarios. The scenarios are represented in sequence diagram where the top rectangles represent objects; the dashed lines connected to objects are the temporal "lifelines"; the arrows between the lifelines represent messages being sent between the objects; and the boxes on the dashed line are activations, showing the execution of a method in response to a message of that object. Figure 4 gives a portion of the sequence diagram for the Entry use case, showing the interactions between the components related to this use case and the messages transferred between them. This use case - Entry – to which this scenario belongs is safety-critical because there are

two hazards are associated with it: - Not letting people out when they need to leave and Admitting intruders.

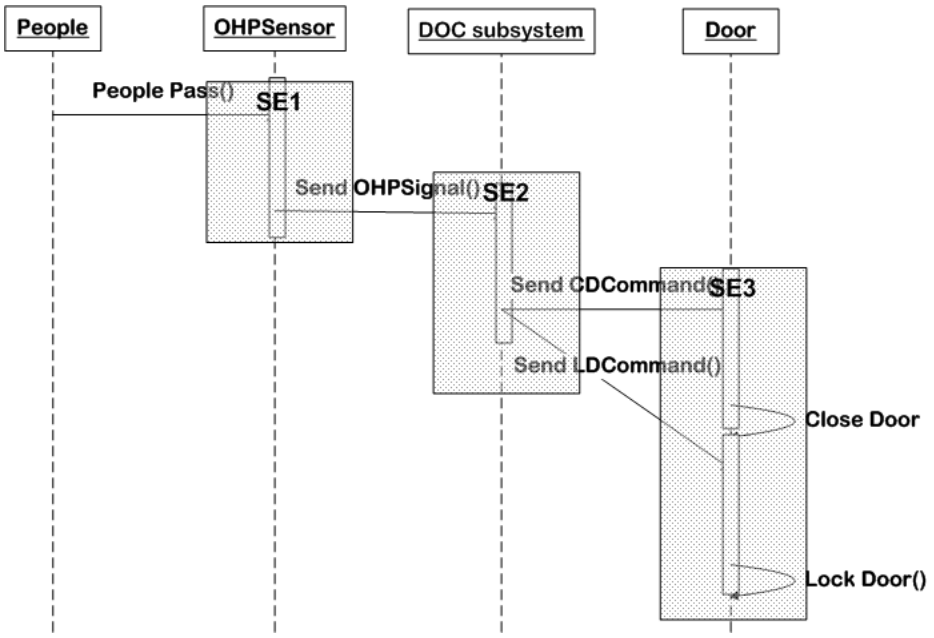

Figure. 4. Sequence diagram for the Close-Door-After-People-Pass scenario

Construction of the sequence diagrams not only made us get more understanding about the system but also identified a missing connector in the architecture diagram. The communication between F_registration detector and RecognitionComponent should be bi-directional, i.e., after the RegistrationComponent component has registered or rejected the ID, it should return a signal to the F_registration detector regarding success or failure.

The construction of the sequence diagram also led to the discovery of a need for additional commands: the Lock Door Command (LDCommand) and the Unlock Door Command (ULDCommand). Initially only an Open Door Command (ODCommand) and a Close Door Command (CDCommand) were used to control the door's opening and closing. Analysis of the Fire Alarm use case showed that to close the door but keep it unlocked in the BedRoomDoor requires two pairs of commands. One command opens or closes the door; the other locks or unlocks the door. To close a door, the DCS needs to send out the CDCommand first, followed by the LDCommand; to open a door, the DCS needs to send out the ULDCommand first, followed by the ODCommand. When the fire alarm is on, the DoorOpenCloseComponent will send out the CDCommand and the ULDCommand to the BedRoomDoor and the SecurityDoor. However, in the FrontDoor, when the fire alarm is on, the DoorOpenCloseComponent will send out the ULDCommand and the ODCommand. The resulting corrections yielded a more accurate foundation for the subsequent Safety Analysis.

4.3. XCA

The Extended Commonality and Variability Analysis (XCA) derived from both the CA and the architecture and sequence diagrams provides the foundation of the product-line safety analysis.

The core architecture design and the events common to shared components give information about the commonalities for the product line, while the differences between individual architectures and the core, as well as the events that are particular to only some systems, give additional information about the variabilities. In the XCA we integrate the core architecture and common events with the commonalities of the product line. The variations of the product architectures from the core architecture and the non-common events are likewise associated with the variabilities of the product line.

First, we obtain events from the sequence diagrams. There are a total of five use cases, each represented by several scenarios. Each scenario is composed of several events. In a sequence diagram, every connection between two actors or devices is an event, representing the sending of a message from a component and its receipt by another component. Every box on a vertical line indicates an internal event, usually the acceptance of a message and the generation of the next message within a component. For example, Figure 4 is a sequence diagram for the scenario in which the door closes after people pass, which is associated with three components: Object-has-passed sensor, DoorOpenCloseComponent (DOC), and Door. The connector between the sensor software and the DOC represents the event of sending a signal; the box in SE2 represents the internal events of generating the CloseDoorCommand (CDCommand) and the LockDoorCommand (LDCommand). For the safety analysis, we operate at a slightly abstract level by combining the receipt of a message, the resulting internal event, and the possible subsequent output of a command as one event. For example the SFMEA analysis considers three events in the sequence diagram in Figure 4 – SE1, SE2, and SE3.

Second, in order to maintain traceability between the events investigated in the safety analysis and the Commonality and Variability Analysis, each commonality in the CA is refined into several sub-commonalties associated with the corresponding events. For example, the commonality C5: "Close door after people pass" can be expanded into three sub-commonalties according to the three events described above. They are:

C5-1: "The event of a person passing through the door will trigger the activity of the Object-has-passed sensors, and the sensors software will send out the OHPSignal to the DOC";

C5-2: "After the DOC accepts the OHPSignal, the DOC will send out the CloseDoorCommand and the LockDoorCommand to the door"; and

C5-3: "After the door has accepted the CloseDoorCommand, it will be closed and after the door has got the LockDoorCommand, its lock will be locked".

C5-1 is a sub-commonality associated with the *component* Object-has-passed-sensors and the *event* that the person has passed by. C5-2 is a sub-commonality associated with the component DOC and the event of accepting OHPSignal, generating CDCommand and LDCommand, and sending out them. C5-3 is a sub-commonality associated with the component Door and the event of accepting CDCommand and LDCommand.

In order to better manage the data, we group the sub-commonalties according to the use-cases to which they belong. For example, all the sub-commonalities expanded from C2: "recognize family members" and C3: "open door for family members from outside" are grouped together, since they are related to the same use case: "Entry".

Similarly, variabilities are refined into sub-variabilities. The method is slightly different from refining commonalities. For each variability we observe from the architectures and the sequence diagrams what the rationale is for the existence of that variability. For example, the variability V3 in the CA is "whether or not the door can be locked from inside". From examination of the individual architectures, we identified the components that allow the door to be locked in some products and not in others, i.e., the existence or non-existence of a door-lock button. From examination of the sequence diagrams, we found that the

sequence diagrams that represent the variabilities have different components. In this example, we found that there are two sub-variabilities that determine the existence of this initial variability, V3. These are : V3-1: "whether or not there is a inside lock door button" and V3-2: "whether or not the DOC component can handle the ButtonPressedSignal".

Third, we construct the Extended Commonality and Variability Analysis (XCA) with the additional architecture and event information needed to support safety analysis. We provide below excerpts from the XCA of the Door Control System.

4.3.1. Terminology

Note that we group the terms according to the different use cases: Registration, Entry, Exit, Fire Alarm, and Bolt Door. Table 2 shows some terms that represent the data (commands and signals) transferred between the components in two representative safety-related situations. The first is a safety-critical use case (Fire Alarm) and the second is a safety-critical scenario (Close Door After People Pass) of another use case (Entry) that also contains non-safety-critical scenarios.

Table 2
Data transferred between components

Data Name	**Data Description**
Situation one: Fire Alarm Use Case	
FAOnSignal	Fire Alarm-is-on signal
FARCommand	Fire Alarm-response command
CDCommand	Close door command
ULDCommand	Unlock door command
ODCommand	Open door command
Situation two: "Close door after people pass" scenario in Entry use case	
OHPSignal	Object-has-passed signal
CDCommand	Close door command
LDCommand	Lock door command

4.3.2. Commonalities

We here list the sub-commonalties for the two safety-critical situations listed above.

Use case 6: Fire alarm (C7: The door will respond to the fire alarm)

C7-1. The doors have a FireAlarmDetector sensor.
C7-2. The FireAlarmDetector will sense when the fire alarm is triggered and will send a signal to the FireAlarmComponent.
C7-3. After the FireAlarmComponent recerives the fire alarm signal, it sends the FireAlarmResponseCommand to the DoorOpenCloseComponent.
C7-4. The DoorOpenCloseComponent will send out the corresponding commands to the door after it has received the FireAlarmResponseCommand.

Use case 2: Entry (C5: Close door after people passing)
(Note that we here list only those commonalities related to the safety-critical scenario "Close door after people pass")

C5-1. The door has a PeoplePassDetector sensor.
C5-2. The PeoplePassDetector will sense when people have cleared the door and then will send out

the ObjectHasPassedSignal to the DoorOpenCloseComponent.

C5-3. After the DoorOpenCloseComponent has accepted the ObjectHasPassedSignal, it will send out the CloseDoorCommand and the LockDoorCommand, respectively and continuously to the door after it gets the ObjectHasPassedSignal.

C5-4. After the door has received the CloseDoorCommand it will be closed and after the door has received the LockDoorCommand its lock will be locked.

4.3.3. Variabilities

We list all the variabilities.

V1-1. The RegistrationDetector is different in different products: F_registration detector (fingerprints) or V_registration detector (voiceprints) or FV_registration detector (both)

V2-1. The RecognitionDetector is different in different products: F_ recognition detector or V_ recognition detector or FV_ recognition detector.

V3-1. Whether or not there is an inside lock door button

V3-2. Whether or not the DOC can handle the ButtonPressedSignal

V4-1. Whether or not the DOC sends out CloseDoorCommand and UnlockDoorCommand, or UnlockDoorCommand and OpenDoorCommand, to the door after it gets FireAlarmResponseCommand

V4-2. Whether or not the door will be open or closed after the fire alarm is on.

V5-1. The OpenFromInsideDetector is different in different products: InsideWeightDetector or InsideVoiceDetector.

V6-1. Whether or not the RecognitionComponent will count the number of illegal IDs that have been input and send out the IllegalInputsignal.

V6-2. Whether or not the IllegalEntryComponent will accept the IllegalInputSignal form RecognitionComponent component.

4.4. Tag the commonalities and variabilities to the architecture diagram

The XCA gives us a clear requirements specification of every component in the DCS. We see that even in this simple product line with only a few variabilities, that the variabilities impose architectural constraints. To verify the completeness of the XCA, we label the connectors in the architecture diagrams with the indices of the associated commonalities and variabilities. Each commonality and variability should associate with one or many connectors in the architecture diagrams. At the same time, every connector in the architecture diagrams should be tagged with one or many commonalities or variabilities. Figure 5 shows an architecture design of the FrontDoor tagged with the commonalities and the variabilities from the safety-critical situation: the Fire Alarm use case and the Close door after people pass scenario.

By associating the components and connectors in the architecture diagrams with the commonalities and variabilities that use them, we capture additional information needed for the safety analysis and establish the architecture as a structuring device to generate the SFTA and SFMEA.

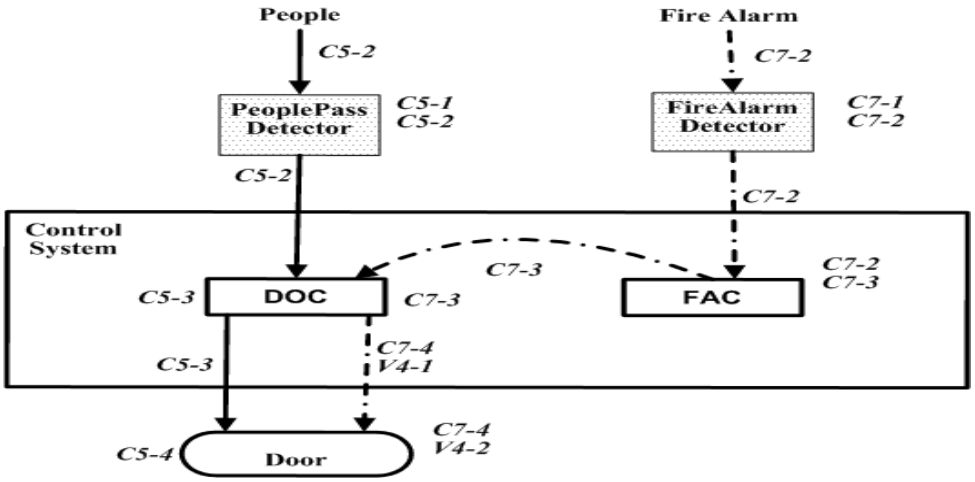

Figure. 5. Architecture of FrontDoor tagged with commonalities and variabilities

5. Software Safety Analysis

In the previous section, we described Step 2 and Step 3 of the product-line safety-analysis process shown in Figure 1. In this section, we discuss Steps 4-6 of this process, namely the production of the safety-analysis artifacts SFTA and SFMEA, and the translation of both into XML.

The method for software safety analysis of a product line proposed here uses the XCA and a hazards list to drive the bi-directional safety analysis. In this section, we first describe the hazard analysis and the backward and forward safety analyses. We then describe their translation into XML files that serve as reusable assets for the product line. Finally we describe the results of applying our method to the Door Control System.

5.1. Hazard Analysis

A detailed discussion of the hazard analysis is beyond the scope of this paper. Briefly, we used a Functional Hazard Assessment (FHA) to identify the following eight hazards [Allenby and Kelly, 2001]:

1) Don't let in the correct people.
2) Let in the wrong people.
3) Don't let people out.
4) When the fire alarm is on, the door doesn't response as it should and people inside cannot get out.
5) The door cannot be locked from the inside, so people inside don't have privacy.
6) The door cannot be opened after it is locked inside, so people cannot get out or get in.

5.2. Backward Safety Analysis – Software Fault Tree Analysis (SFTA)

SFTA is a "top-down" safety analysis technique that is used to identify the errors, faults and failures that could contribute to hazards. It is a means for analyzing causes of hazards, which are identified as the top events. A backward search from the top event is performed to find out combinations of contributing causes of the hazards [Leveson, 1995; Lutz and Woodhouse, 1997]. The tree is equivalent to a predicate statement with the nodes connected by AND and OR gates. The SFTA is guided by the XCA.

Every intermediate node is decomposed into sub-trees. The sub-trees contribute to the occurrence of

the parent. Combination errors include the errors caused by the occurrence of two or more correct events at the same time, errors caused by the race conditions of two or more events, errors caused by the occurrences of two or more errors at the same time, and errors caused by the handling of two or more errors at the same time. Every error is decomposed into sub-trees until the leaf nodes are reached. The leaf nodes are analyzed using the failure modes from SFMEA, such as "absent data", "incorrect data", "wrong timing of data", "duplicate data", "halt/abnormal termination of event", "omission", "incorrect logic/event", and "timing/order", [Lutz and Woodhouse, 1997]. We considered if those failure modes of the data and events that are associated with each component could contribute to the errors of the component that were on the bottom of the fault tree.

We catalog the fault tree's nodes into five types: "Hazard", "Intermediate", "Leaf", "ProductsDivision", and "Reused". The root node is level zero; a hazard node is at level one of the fault tree. An intermediate node is a node that is not a leaf, a hazard, a reused node, or a ProductsDivision node. A leaf node is a node that is at the bottom level of the fault tree. A reused node is a shorthand specification of a sub-tree that repeatedly occurs. A ProductsDivision node is a node related to the divisions of the tree associated with different products. ProductsDivision nodes provide an abstraction of the concrete errors of different products in the nodes of the next level. The advantages of these nodes are the following:

- Understandability. With these kinds of nodes, we can easily locate the place that the variations occur.
- Reuse and evolution. The sub-tree will be a complete tree for several products with the same variability. In this way, we can easily edit and trim the fault tree [Dehlinger and Lutz, 2004].

Since every node in the fault tree can be connected to a product or a group (subfamily) of products, we tag every node in the fault tree with the appropriate indices of the XCA, so that every node in the fault tree is associated with at least one commonality or one variability. The reason for the tagging is to make the product line's fault tree easy to understand and easy to edit and prune in the future. For example, suppose we want to derive a new product with the additional privacy feature of being able to lock the FrontDoor from inside against even people with valid access. In this case, we can check the nodes' tags to identify all the nodes marked with the tags of the commonalities and variabilities related to Lock-Door-Inside, Fingerprint-Registration and Fingerprint-Recognition.

We have created an entire fault tree for the Door Control System product line. The root node is "Malfunction of the Door Control System" and the nodes in level one are the six hazards described above. The nodes in level two are the errors and faults that can contribute to the hazards, etc. We omit the fault tree here for space reasons but include pieces of its XML representation below.

5.3. Forward Safety Analysis – Software Failure Mode and Effect Analysis

The Software Failure Mode and Effect Analysis (SFMEA) is a bottom-up forward search from failure modes associated with data and events to the effects that are caused by those failure modes. [Leveson, 1995; Lutz and Woodhouse, 1997]. The first step is to identify all the components in the system, which we have already done through the architecture diagrams. Lutz and Woodhouse [Lutz and Woodhouse, 1997] provide a list of generic failure modes: four associated with data communication and four associated with event processing. The four failure modes for data are "incorrect value", "absent value", "wrong timing", and "duplicated value". The four failure modes for events are "halt/abnormal termination", "omission", "incorrect logic/event", and "timing/order". The events are obtained from the sequence diagrams as described in Section 4.2. The data are obtained from the architecture diagrams and the sequence diagrams. In the architecture diagrams, the communication data is the data transferred along the connectors. In the sequence diagrams, the data are transferred between two vertical lines

The safety analysis groups the data according to the use cases. For example, in the Fire Alarm use case group, the data include FireAlarmOnSignal, FireAlarmResponseCommand, CloseDoorCommand, UnlockDoorCommand, and OpenDoorCommand. Each data item is essential to make this use case occur successfully and no other data is necessary to its occurrence.

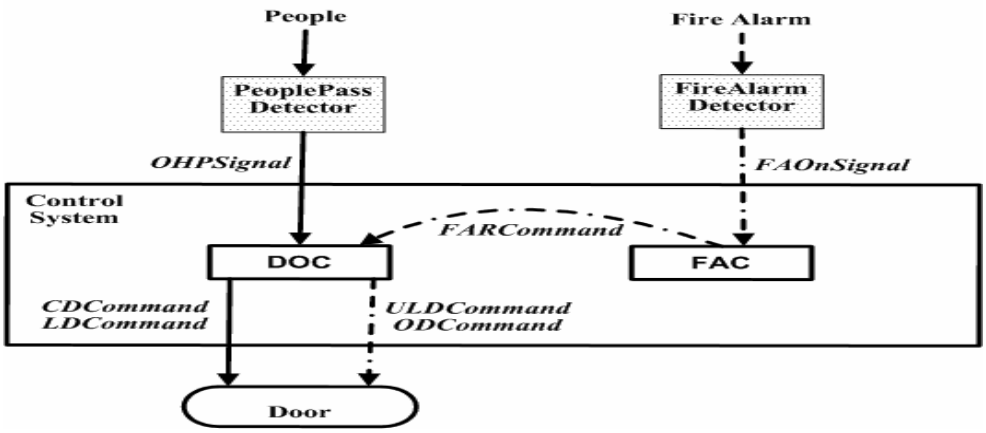

Figure. 6. Mapping data to the architectural design

Figure 6 shows the relationship between the use case data and the architecture for the use case: Fire Alarm use case and the portion of the Entry use case: Close-Door-After-People-Pass scenario. By tagging the SFMEA's data with the architectural connectors, we ensure coverage of the data that communicate among components. That is, the SFMEA depends on the architecture diagrams. However, note that different use cases may use the same data. For example, the CloseDoorCommand is in both the Fire Alarm use case and in the Entry use case. However, since both the reasons for the generation of these data and the hazards to which they can contribute may differ, we treat data with the same name but from different use cases as distinct from each other.

The schema of our SFMEA data table includes "Index", "Data Item", "Group", "Product", "Data Failure Mode", "Description", "Local Effect", "End Effect", and "Possible Hazard", as shown in Table 3.

Note that in every row in the data table, a failure mode of a datum is identified and we assume that every other datum is not in any failure mode. In reality, this is not always true. For example, a failure of the OpenDoorCommand is probably always associated with a failure of the UnlockDoorCommand. However, the SFMEA's event table does consider such combinations of errors, as long as these data are in the same event.

Table 3
Schema of SFMEA's data table

Data Item	Group	Product	Data Failure Mode	Local Effect	End Effect (System effect)	Possible Hazard
The name of the datum.	The name of the group that the datum belongs to.	The products' name that the datum is associated.	Incorrect value, absent value, wrong time, or duplicated value.	The effect that occurs to the components that the datum is directly associated with if the datum is in the failure mode.	The system failure if the local effect occurs.	The possible hazard that happens if the end effect occurs.

Table 4 is excerpted from the SFMEA for the OpenDoorCommand in the Fire Alarm use case. In our example, we only include one failure effect for each failure mode, although there are several effects associated with some failure modes.

Table 4
SFMEA on the OpenDoorCommand of Fire Alarm group

Data Item	Group	Product	Data Fault type	Local Effect	End Effect	Possible Hazard
OpenDoor Command	Fire Alarm	FrontDoor	Incorrect Value	The DoorOpenClose component does not receive FireAlarmResponseCommand, but the DoorOpenClose sends out OpenDoorCommand.	When the fire alarm is not on, the door is incorrectly opened.	Lets in the wrong people.
			Absent Value	The DoorOpenClose component receives FireAlarmResponseCommand, but the DoorOpenClose does not send out OpenDoorCommand.	When the fire alarm is on, the door is incorrectly not opened.	The fire alarm is on and people cannot exit.
			Wrong timing	The DoorOpenClose component receives the FireAlarmResponseCommand, but the OpenDoorCommand is delayed for at least 1 min.	When the fire alarm is on, the door's opening has been delayed for at least for 1 min.	The fire alarm is on and people cannot exit quickly.
			Duplicate Value	The DoorOpenClose component receives the FireAlarmResponseCommand, and then the DoorOpenClose component sends out two OpenDoorCommand.	When the fire alarm is not on, the door is incorrectly opened.	Let in the wrong people.

The schema of the event table of SFMEA is similar to the data table's. Table 5 shows a SFMEA on an event for the FrontDoor product in the Fire Alarm use case. In this event, the DoorOpenCloseComponent receives the FireAlarmResponseCommand, and then generates and sends an UnlockDoorCommand and an OpenDoorCommand to the Door. Again, we here show only one failure effect per failure mode.

As in the data table, although the events table may have two or more events with different names doing the same thing, we still name them differently because their triggers and potential hazards may be different.

Table 5
SFMEA on an event for the FrontDoor product in the Fire Alarm use case

Group	Product	Event Fault type	Local Effect	End Effect	Possible Hazard
Fire Alarm	FrontDoor	Halt/Abnormal termination	The DoorOpenCloseComponent receives the FireAlarmResponseCommand,	The fire alarm is on, but the door is incorrectly not	The fire alarm is on and the people cannot exit.

			but it does not send out the UnlockDoorCommand and the OpenDoorCommand to the door.	opened.	
		Omission	The DoorOpenCloseComponent receives the FireAlarmResponseCommand, but it does not send out the UnlockDoorCommand and the OpenDoorCommand to the door.	When the fire alarm is on, the door is incorrectly not opened.	The fire alarm is on and the people cannot exit.
		Incorrect logic/event	The DoorOpenCloseComponent does not receive FireAlarmResponseCommand. But it sends out the OpenDoorCommand or UnlockDoorCommand or both.	When the fire alarm is not on, the door is incorrectly opened.	May let the wrong people in.
		Timing/Order	The DOC component receives the FireAlarmResponseCommand, but sends the UnlockDoorCommand later than the OpenDoorCommand.	The fire alarm is on. The door is unlocked, but it's incorrectly not opened.	When the fire alarm is on, people cannot exit.

5.4. Convert SFTA and SFMEA to XML files

In order to make the safety analyses available and reusable when adding new products to the product line, we convert the results of the SFTA and the SFMEA into XML files in a partially-automated step. The reasons for selecting XML in this research (e.g., instead of a relational database) are as follows:

Basically, the advantage of XML is that since the output of the SFTA is a tree, it can be readily translated into XML by the FaultCat tool. Furthermore, Finally, XML is as easy to manipulate as a relational database and as powerful for our purposes.

1. The result of the SFTA is a fault tree, which is stored as an XML file by the FaultCat tool used to construct the fault tree.
2. The representation of fault trees in XML is becoming fairly standard. For example, the Galileo fault tree tool at the University of Virginia uses XML [Sabanosh and Sullivan, 2001], as do commercial fault-tree tools, such as Relex. XML will likely become even more widely accepted in industry as the exchange language for analysis toolsets as XML parsers are built into programming languages.
3. XML representations of fault trees are easy to understand and manipulate, so appeal to software developers. For example, the correspondence between the graphical representation of nodes and their hierarchy in the XML nodes is clear without training.

The resulting XML files also allow automated comparison of the two safety analyses. In addition, a product line can be expected to evolve over time. The XML representation can be easily edited to incorporate changes.

For example, to build a fault tree for just the product BedRoomDoor, we can partial-automatically extract those nodes belonging to the BedRoomDoor to build a new fault tree for that product. In this way, the reuse of safety analysis of a product line can be achieved more accurately and quickly.

```
<!ELEMENT fault-tree ( node+ ) >
<!ELEMENT node (Name, Parent, Gate, Type, Products, ComOrVar, Description, Children) >
<!ELEMENT Products (Product +) >
<!ELEMENT Description (Content, KeywordsSet) >
<!ELEMENT KeywordsSet (Keyword +) >
<!ELEMENT Children (ChildrenNum, Child +) >
<!ELEMENT Gate ( “” | “Or”| “And” ) >
<!ELEMENT Type ( “Hazard” | “Error” | “ProductsDivision” | “Reused” ) >
```

Figure. 7. Wanted XML DTD elements

```
<Node>
   <Parent>B4</Parent>
   <Name>B38</Name>
   <Type>Error</Type>
   <Gate>Or</Gate>
   <ComOrVar>
      <Name>Com</Name>
      <Products>
         <Product>P_F</Product>
         <Product>P_B</Product>
      </Products>
   </ComOrVar>
   <Description>
      <Content>The DOCSub does not accept OHPSignal. But it sends out the ODCommand. So the
            door is closed before the correct people's passing</Content>
      <KeywordSet>
         <Keyword>DOCSub</Keyword>
         <Keyword>does not</Keyword>
         <Keyword>accept</Keyword>
         <Keyword>OHPSignal</Keyword>
         <Keyword>CONDITION</Keyword>
         <Keyword>DOCSub</Keyword>
         <Keyword>sends out</Keyword>
         <Keyword>ODCommand</Keyword>
      </KeywordSet>
   </Description>
   <Children>
      <ChildrenNum>0</ChildrenNum>
   </Children>
</Node>
```

Figure. 8. A node of the SFTA

We construct the fault tree using the Fault Tree Creation & Analysis Tool (FaultCAT) [Burgess, 2003], which allows us to draw and edit the fault tree and convert it into an XML file. Because we want every node in the fault tree to be an independent node (somewhat similar to [Sabanosh and Sullivan, 2001]), we wrote a Java program, which takes as input the original XML file with the format from FaultCAT and writes out an XML file with the desired format as shown in Figure 7 in DTD format [DTD tutorial]. The advantage of our format is that it is easy to understand and easily program readable. Figure 8 is an example of a leaf node of the Fault Tree in our format.

For the SFMEA table each data or event is converted into one node of the XML file. Figure 8 gives a description of the XML file of the SFMEA’s data table, using DTD [DTD tutorial]. We wrote a Java

program to transfer the tables of the SFMEA to a XML file using the Java Excel API.

```
<!ELEMENT SFMEA_data ( Data + ) >
<!ELEMENT Data (NodeIndex, DataIndex, DataName, DataNameLong, Products, Errors)>
<!ELEMENT Errors (OneError +) >
<!ELEMENT OneError (OneErrorIndex, DataFailureMode, Description, LocalEffect, EndEffect, PossibleHazard)>
<!ELEMENT LocalEffect (Reused | (LocalEffectContent, KeywordsSet) >
<!ELEMENT EndEffect (Reused | (EndEffectContent, KeywordsSet) ) >
<!ELEMENT KeywordsSet (Keyword+) >
```

Figure. 9. DTD for SFMEA

By partically automated, we mean that the Java programs to translate the XML files and the Xlinkit rules to compare the XML files only have to be written once. Subsequently, the programs automatically compare and check the files. The advantage of having partially automated comparison is that it is significantly faster and lower-cost than manual comparison.

6. Results and Evaluation

This section describes Step 7 of the safety-analysis process for product lines that was summarized in Figure 1. The section presents and evaluates the results both in terms of the automated consistency checking of the forward vs. backward safety analysis results and by discussing the missing software safety requirements for the product line found by this method.

6.1. Consistency Checking of SFTA and SFMEA

To evaluate the consistency and completeness of the safety analyses, we use patial automation to compare the SFTA and SFMEA XML files. The tool that we used to do the comparison of the keywords set, called "Xlinkit", can apply rules on multiple XML documents [Nentwich, 2002; Nentwich et al., 2002]. To accomplish this we conducted a comparison of SFTA and SFMEA, i.e., comparing every node of the fault tree with every failure effect of the SFMEA and vice versa. The method of comparison is using the keywords set and Xlinkit. Figure 10 shows a portion of the code for the Xlinkit rule that checks if the fault tree's nodes are in the SFMEA's effects. For every node in the fault tree for which there exists the same failure effect in the SFMEA data table, the node's keywords must exist in the keywords of that failure failure mode's failure effect and vice versa.

A keyword of a phrase is a word that is necessary to express the correct meaning of this phrase. So, to compare two phrases, we can compare two keyword sets. Our comparison of the SFTA and the SFMEA was based on the assumption that the architecture design used in SFTA and the one used in SFMEA are at the same level of detail. More precisely, if the architecture design used in SFTA has a component A, then the architecture design used in SFMEA has the same component A; if this component A has n number of sub-components in the design used in SFTA, then the component A has the same number of sub-components in the design used in SFMEA.

We compared the *faults* of SFTA and the *failure effects* from SFMEA from the Fire Alarm use case and the "Close-Door-After-People-Pass" safety-critical scenario from the Entry use case. There were thirty-eight nodes in the SFTA's Fire Alarm use case and ten nodes in the SFTA's Close-Door-After-People-Pass scenario. In the corresponding SFMEAs, there were eight data items analyzed and thirty-two failure effects; nine events analyzed and seventy-five failure effects. Note that between the data tables and the event tables, there were many redundant failure effects, i.e., some failure effects appeared

repeatedly. Within failure modes, a small set was identical and some had the same meaning.

```
<or>
        <exists var="oneDataError" in="/SFMEAData/Data/Errors/OneError">
           <or>
                <and>
                        <forall var="oneSFTKeyword" in="$oneNode/Description/KeywordSet/Keyword">
                                <exists var="oneDataLocalKeyword"
                        in="$oneDataError/LocalEffect/KeywordsSet/Keyword">
                                        <equal op1="$oneDataLocalKeyword/text()"
                                op2="$oneSFTKeyword/text()" />
                                 </exists>
                         </forall>
                        <forall var="oneDataLocalKeyword"
                in="$oneDataError/LocalEffect/KeywordsSet/Keyword">
                                <exists var="oneSFTKeyword"
                        in="$oneNode/Description/KeywordSet/Keyword">
                                        <equal op1="$oneDataLocalKeyword/text()"
                                op2="$oneSFTKeyword/text()" />
                                 </exists>
                          </forall>
                  </and>
                  <and>
                        …
                  </and>
                        …
```

Figure. 10. A part of a rule of Xlinkit

From the comparison using Xlinkit, we found that the SFMEA is more complete than the SFTA when both of them were at the same level of detail. Here, the same level of detail refers to the architecture level as described above. Seventeen nodes in the fault tree with the type “Error” and with the gate “Or” or “And”, were not in the SFMEA data table or in the SFMEA event table. After checking the seventeen nodes manually, we found that, in fact, fifteen nodes do have corresponding effects in the SFMEA. One reason that Xlinkit could not match them is because they had different keywords. For example, node B11 in the SFTA was not found in the SFMEA. The KeywordSet of the B11 node in the fault tree included the keywords: “door”, “does not”, “accept”, “CDCommand”, “door”, “close”, “without”, “people”, and “pass”. However, after checking manually, we found a failure mode in the SFMEA with the KeywordSet containing “door”, “does not”, “accept”, “CDCommand”, “ULDCommand”, “door”, “close”, and “unlock”. Although these two KeywordSets were different, they actually described the same fault. Both relate to the undesirable behavior of the door closing automatically without having been commanded to close. Another reason that Xlinkit could not match some KeywordSets is that in one case, the analyst had inadvertertently omitted a pre-condition from the SFTA but included it in the SFMEA.

We also checked whether the failure effects from the SFMEA matched the faults of the SFTA. Again we found a significant number of initial mismatches (sixteen failure effects in the event tables that were not found in the SFTA and twelve failure effects in the data table that were not found in the SFTA,

excluding timing errors). We manually checked to see why those effects were not in the SFTA. Half of the mismatches were due to the keywords sets' variability. The others were due to the SFMEA being more detailed than the SFTA. For example, only the SFMEA considered the possibility that LockDoorCommand is sent while the CloseDoorCommand is not when the door is closed. The SFTA considers the two commands together.

6.2. Identification of New Safety Requirements

The application of the safety analysis methodology to the Door Control System product line identified some missing safety requirements. Some new requirements were commonalities that addressed shared hazards. Other new requirements affected only some products in the product line (i.e., were variabilities).

An example of a new shared safety requirement came from the forward search that showed that if the digitalized fingerprint image data were in the failure mode "incorrect value", this could have the effect of letting in intruders. Performing a backward search using the failure mode "incorrect value" as the root to find out why it failed to give the correct fingerprint images identified the possible cause that the F_ recognition detector does not accept the new input and continues to send out the old data. This finding resulted in a new safety-related requirement that sensors must purge old data, thus also requiring the addition of an expiration time for all data that is used in a control decision.

We also found a new safety requirement by inspection of the architecture for race conditions. In every product, if a person has passed the door, the Object-has-passed sensors will send out the ObjectHasPassedSignal to the DoorOpenClose component, which will then send out a CloseDoorCommand and an LockDoorCommand, respectively. This is a correct operation of the Door Control System. However, in the FrontDoor product, when the fire alarm is on, the FireAlarmDetector will send out the FireAlarmOnSignal to the FireAlarmComponent, which will send out the FireAlarmResponseCommand to the DoorOpenCloseComponent. The DoorOpenCloseComponent will then send out an UnlockDoorCommand and an OpenDoorCommand to the door.

A race condition can occur if at the moment that a person passes through the door, the fire alarm is on. If the UnlockDoorCommand and OpenDoorCommand get to the door later, then there is no hazard. However, if the CloseDoorCommand and the LockDoorCommand get to the door later, then the door will be closed and locked while the fire alarm is on. In the product BedRoomDoor, the same race condition can also cause this dangerous behavior when the fire alarm is on.

To avoid this race condition, a new safety-related behavioral commonality was added to the system: after the DoorOpenCloseComponent receives the FireAlarmResponseCommand, it will not accept any other commands until the fire alarm is off. This also results in a new requirement for additional functionality to move the DoorOpenCloseComponent out of the frozen state.

Other instances of incompleteness in the product line requirements found by the bi-directional safety analyses were that there should be sensors on the edges of the doors to prevent users from being pinned and that there should be software time-outs to control the door's opening and closing. We also found a missing variability for the BedRoomDoor and the SecurityDoor, i.e., that when the fire alarm is on, the door must be unlocked and closed, even if the door's initial state is locked inside. Through the safety analysis (specifically the SFMEA) we also discovered a new possible hazard: "people are pinned between the doors," common to all the products in the Door Control System product line.

7. Conclusion and Future Work

This paper has presented a method for performing safety analysis on a software product line and demonstrated the method on three members of a safety-critical product-line, the Door Control System for a SafeHome application subsystem. The work described here extended the product-line commonality and variability analysis with domain information from the product-line architecture and sequence diagrams. The resulting representation, called the Extended Commonality Analysis, was then used to guide the bi-directional safety analysis. The intermediate products of the bi-directional safety analysis, SFTA and SFMEA, were converted to XML files and, using rules coded by the analyst, automatically compared via the software package, xlinkit. Omissions and inconsistencies were then identified and removed, providing a more-thorough safety analysis. Making the safety analyses available to the projects as XML files provides an important first step toward partially automated updating of safety analyses as requirements evolve, and toward reuse of the safety analyses in the application-engineering phase as new systems are built.

Findings from application of the bi-directional safety-analysis method included new safety-related software requirements both for all the systems in the product line (commonalities) and for only some of the product-line systems (variabilities), as well as discovery of a new hazard, that people can be pinned by the door. The paper provides a structured method and step-by-step guidelines for deriving a safety analysis from an extended commonality analysis in order to improve the safety of the software product line. The method is general, so can help assure the safety of other critical product lines. To this end, we are currently investigating the scalability and domain-independence of the method in an application to a real-world industrial product line.

Acknowledgements

The authors would like to thank Josh Dehlinger, Oko Swai, and Jing Liu for their comments and useful discussions. We also thank Anthony Finkelstein and Christian Nentwich for the use of xlinkit. This research was supported in part by National Science Foundation grants 0204139 and 0205588.

References

Allenby, K., Kelly, Trim., 2001. Deriving Safety Requirements Using Scenarios: Requirements Engineering. In: Proceedings of the Fifth IEEE International Symposium, Toronto, Ont., Canada, pp. 228-235.

Ardis, A. M., Weiss, D. M., 1997. Defining Families: The Commonality Analysis, Software Engineering. In: Proceedings of the 1997 (19th) International Conference, pp. 649–650.

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J., Zettel, J., 2002. Component-based Product Line Engineering with UML, Addison-Wesley, 2002.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, Addison-Wesley.

Bayer, J., Flege, O., Gacek, C., 2000. Creating Product Line Architectures. In: Proceedings of the 3rd International Workshop on Software Architectures for Product Families. Lecture Notes in Computer Science Volume 1951, Springer-Verlag, pp. 210-216.

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K, Widen, T., 1999. PuLSE: A methodology to develop software product lines. In: Proceedings of the 5th Symposium on Software Reusability, Los Angeles, CA, USA, pp. 122-131.

Bosch, J., 2000. Design and Use of Software Architectures. Addison-Wesley.

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

Clauß, M 2001. Modeling variability with UML. In GCSE 2001Young Researchers Workshop, Erfurt, Germany.

Cook, D. J., Huber, M., Gopalratnam, K., Youngblood, M., Learning to Control a Smart Home Environment. Available from http://ranger.uta.edu/~holder/courses/cse6362/pubs/Cook03.pdf. (Accessed July 2004).

Dehlinger, J., Lutz, R., 2004. Software Fault Tree Analysis for Product Lines. In: Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering, Tampa, FL, USA, pp. 12-21.

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

DTD tutorial. Availabel from http://www.w3schools.com/dtd/. (Accessed July 2004)

Egyed, A., Mehta, N. R., Medvidovic, N., 2000. Software Connectors and Refinement in Family Architectures. In: Proceedings of the Third International Workshop on Development and Evolution of Software Architectures for Product Families (ARES III). Lecture Notes in Computer Science Volume 1951, Springer-Verlag, pages 96-105.

Fellbaum, K., Hampicke, M., Human-Computer Interaction in a Smart Home Environment. Available from http://www.sentha.tu-berlin.de/paper/fm_ha_miami.pdf. (Accessed July 2004)

Goseva-Popstojanova, K., Hassan, A., Guedem, A., Abdelmoez, D. W., Ammar, H., 2003. Architectural-Level Risk Analysis using UML. IEEE Transactions on Software Engineering, pp. 946-960.

John, I., Muthig, D., 2002. Tailoring Use Cases for Product Line Modeling. In: Proceedings of International Workshop on Requirements Engineering for Product Lines, Essen, Germany, pp. 26-32.

Kang, K. C., Kim, S., Lee, J, Kim, K., Shin, E., Huh, M., 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering, Volume 5, 1998, pp. 143-168.

Leveson, N. G., 1995. Software System Safety. Addison-Wesley.

Lu, D., Lutz, R., 2002. Fault Contribution Trees for Product Families. In: Proceedings of the 13th International Symposium on Software Reliability Engineering, pp. 231-242

Lutz, R. R., 2000. Extending the Product line Approach to Support Safe Reuse. The Journal of Systems and Software, Vol. 53, Issue 3, pp. 207-217.

Lutz, R. R., Gannod, G. C., 2003. Analysis of a Software Product Line Architecture: An Experience Report. The Journal of Systems and Software, vol. 66: 3, pp. 253-67

Lutz, R. R., Woodhouse, R. M., 1997. Requirements analysis using forward and backward search. Annals of Software Engineering, Vol 3, pp. 459 –475.

MavHome. Available from http://mavhome.uta.edu/. (Accessed July 2004).

Nentwich, C., 2002. Xlinkit Version 5 Language Reference. Available from http://www.systemwire.com/xlinkit/Downloads/xlinkit-5.3/doc/reference/reference.html. (Accessed July 2004).

Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A. F., 2002. xlinkit: A Consistency Checking and Smart link Generation Service. ACM Transactions on Internet Technology, Vol. 2, pp. 151-185.

Perry, D. E., 1998. Generic architecture descriptions for product lines. In: Proceeding Of ARES II: Software Architectures for Product Families (LNCS 1429), Springer-Verlag, pp. 51-56.

Sabanosh, T. Sullivan, K., 2001. Interoperability for Probabilistic Risk Assessment Tools Using Domain-Specific Markup Languages. Available from http://dependability.cs.virginia.edu/bibliography/Interoperable.pdf. (Accessed July 2004)

Smart Home–project. Institute Electronics at Tamperre University of Technology , Finland. Available from http://www.ele.tut.fi/research/personalelectronics/projects/smart_home.htm. (Accessed July 2004)

Sommerville, I. 2004. Software Engineering. Addison-Wesley.

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

Youngblood, G. M., 2003. Mavhome Research Scope. Available from http://www.aimachines.com/youngblood/papers/MVH-2003-2_MavHome_Scope_2003.pdf. (Accessed July 2004)

Robyn R. Lutz is an Associate Professor in the Department of Computer Science at Iowa State University and a Senior Engineer at the Jet Propulsion Laboratory, California Institute of Technology. She received a Ph.D. from the University of Kansas. Her research interests include software safety, safety-critical product lines, defect analysis, and the specification and verification of requirements, especially for fault monitoring and recovery. Her research is supported by NASA and by the National Science Foundation. She is a member of IEEE and of the IEEE Computer Society.

Qian Feng is currently a graduate student in the Department of Computer Science at Iowa State University, Ames, IA. She received a B.A. in computer science from Iowa State University. Her research interests include software safety, product line engineering, software architecture, and human-computer interaction. Her research is supported by the National Science Foundation.