Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1879–1892
The Journal of Systems and Software
www.elsevier.com/locate/jss

Safety analysis of software product lines using state-based modeling ☆

Jing Liu [a], Josh Dehlinger [a], Robyn Lutz [a,b,*]

[a] Department of Computer Science, Iowa State University, Ames, IA 50011, United States
[b] Jet Propulsion Laboratory/Caltech, United States

Received 15 December 2005; received in revised form 5 January 2007; accepted 8 January 2007
Available online 14 February 2007

Abstract

The difficulty of managing variations and their potential interactions across an entire product line currently hinders safety analysis in safety-critical, software product lines. The work described here contributes to a solution by integrating product-line safety analysis with model-based development. This approach provides a structured way to construct state-based models of a product line having significant, safety-related variations and to systematically explore the relationships between behavioral variations and potential hazardous states through scenario-guided executions of the state model over the variations. The paper uses a product line of safety-critical medical devices to demonstrate and evaluate the technique and results.

Keywords: Product lines; Safety-critical systems; Model-based development; State-based modeling

1. Introduction

The analysis and management of variations (such as optional features) are central to the development of safety-critical, software product lines. We define a software variation as "the ability of a software system or artifact to be changed, customized, or configured for use in a particular context" (Clements and Northrop, 2001). In safety-critical product lines such as pacemakers (Ellenbogen and Wood, 2002), mobile communication devices for emergency workers (Doerr, 2002), constellations of satellites (Dehlinger and Lutz, 2005), and medical-imaging systems (Schwanke and Lutz, 2004), balancing safety assurance and reuse management has become a major obstacle to safety analysis: A safety-critical product line must satisfy its safety properties in all allowable configurations (i.e., choices of variations). The notion of mandatory features (commonalities) and optional variations (variabilities) makes it possible to reuse some analyses of feature interactions. However, they introduce new dependencies (constraints between commonalities and variabilities and among variabilities) that can make it difficult to provide assurance that safety properties will be satisfied in the presence of variations' interactions. This discourages the addition of variations and limits the potential for reuse of product-line artifacts. Therefore, the development of reusable safety analysis techniques that can incorporate the variations without compromising the safety of individual products is needed.

The development of software product lines thus lacks comprehensive methods to ensure the satisfaction of safety properties of the product line while still taking advantage of the product line's inherent reuse potential (Lutz, 2000a,b). Specifically, Kang (2006) identifies the following as open problems for the practical use of product lines for safety-critical software:

- Verifying quality attributes, such as safety and reliability, and detecting feature interactions that may violate the safety properties or quality attributes.

☆ An early version of this paper was presented at ISSRE 2005.

* Corresponding author. Address: Department of Computer Science, Iowa State University, Ames, IA 50011, United States.
E-mail addresses: janetlj@cs.iastate.edu (J. Liu), dehlinge@cs.iastate.edu (J. Dehlinger), rlutz@cs.iastate.edu (R. Lutz).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.01.047

- Modeling, analyzing and managing product-line features and feature interactions while avoiding the feature explosion problem.

The work described in this paper is motivated by and addresses these problems in terms of the design, analysis and development of a safety-critical, software product line.

The state-based modeling approach in this paper builds upon two existing product-line safety analysis techniques: Software Failure Modes, Effects and Criticality Analysis (SFMECA) and Software Fault Tree Analysis (SFTA). The advantages of our state-based modeling approach for the safety analysis of a product line, compared with SFMECA and SFTA, include (Liu et al., in press):

- Analysis and modeling of timing/ordering-sensitive failure events to determine their possible safety implications.
- Simulation of the behaviors described by the requirements in the fault tree, to illustrate the violation of a safety property.
- Exploration of possible solutions when safety properties are found to be violated to identify an adequate mitigation strategy.

For these reasons, chaining the traditional safety analysis mechanisms (SFMECA and SFTA) with the state-based modeling for the product line strengthens the safety analysis across a product line.

The main contribution of this paper is thus a technique to perform safety analysis on the variations in a product line using state-based modeling. This technique makes it more practical to check that safety properties for the product line hold in the presence of variations through scenario-guided execution, or animation, of the model. Further, utilizing the technique described in this paper at the design level allows safety engineers to discover faults early enough to design mitigation strategies before implementation and deployment. Relationships between the behavioral variations and hazardous states can be at that point systematically explored and new safety requirements derived. The improved management and analysis of variations obtained by using this technique promotes safer reuse of artifacts developed for the product line.

The work presented here is part of a larger effort (described in Section 2) to investigate how safety analysis can become a reusable asset of a product line by developing a framework and a suite of techniques for the safety analysis of product lines. The long-term goal is to be able to provide verification results for a new system in the product line in a timely and cost-efficient manner.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 gives an overview of the approach and introduces the running example (a pacemaker product line). Section 4 describes the state-based modeling and execution of the product line to validate safety properties. Section 5 discusses and evaluates the results in terms of their use in safety-critical systems. Section 6 offers some concluding remarks.

2. Background and related work

Our work is based on the overlapping areas of software product-line engineering, model-based development and safety analysis. In comparison with previous work, we focus on the use of state-based models to enable verification of safety-related properties in the presence of product-line variations.

A software product line is a set of software systems developed by a single company that share a common set of core requirements yet differ amongst each other according to a set of allowable variations (Clements and Northrop, 2001; Weiss and Lai, 1999). The product-line engineering methodology is advantageous in that it exploits the potential for reuse in the analysis and development of the core and variable requirements in each member of the product line (Lutz, 2000a). The initial stage of product-line engineering, domain engineering, defines the commonality and variations of the product line. The later stage, application engineering, then binds the variations to specific products (Svahnberg et al., 2005).

Model-based development of critical systems has demonstrated advantages. By executing or animating the model at design time, the sufficiency and correctness of the requirements and design can be verified prior to implementation. State-based modeling has been shown to support verification of behavioral requirements (Czerny and Heimdahl, 1998). Campbell et al. created UML diagrams and then modified the initial values in UML diagrams to create hazardous situations to confirm that the model was still safe (Campbell et al., 2002). Executable UML allows animation of scenarios to verify the models that have been built (Mellor and Balcer, 2002). More recently, Gomaa has shown how executable UML statecharts can be used for product-line models (Gomaa, 2005).

While work to date concentrates on how to validate whether a single system's model behaves correctly, the work reported here investigates whether the members of a product line display safe behavior (i.e., satisfy certain, selected safety properties). The representation of variations in the model is thus of primary importance.

The state-based modeling and safety-analysis method proposed here is compatible with several widely-used product-line engineering frameworks. A variety of feature diagrams or variation models, including those in Kobra (Atkinson et al., 2002), FORM (Kang et al., 1998) and FAST (Weiss and Lai, 1999), can be annotated with references to the associated modeling elements in this work. Sophisticated tool support, such as the Rhapsody tool-set we used here, allows the animation of a state-based design model (Douglass, 1999). Similarly, our use of scenario-guided testing of state-based design models has been previously shown to provide a powerful way to identify omissions in requirements (Harel and Marelly, 2003).

There has been widespread attention to UML modeling of product lines (Gomaa, 2005). From the perspective of promoting reusable components, Clauss extends UML to support feature diagrams and adds elements describing variations to the UML package diagram (Clauss, 2001). Doerr categorizes the relationships that exist in a variation model and also ties these to UML notation (Doerr, 2002).

Safety analysis for software product lines is still in its infancy. As in a single system, the derivation of safety properties comes from the hazard analysis (Leveson, 1995) for a product line. Previous work has described a forward search for hazardous effects of possible failures coupled with a backward search from faults to their contributing causes as a means for safety analysis of a single system (Lutz and Woodhouse, 1999).

More recently, we have worked to detail how safety analysis assets can be reused within a product line as well as to identify specified limits on reuse of product-line safety analysis assets. We have provided an extension of bi-directional, safety analysis approach to product lines (Feng and Lutz, 2005), used it to aid in the certification of safety-critical product lines (Dehlinger and Lutz, 2005), and produced product-line analysis tools (e.g., PLFaultCAT (Dehlinger and Lutz, 2006), DECIMAL (Padmanabhan and Lutz, 2005)). The techniques and tools in this framework aim to provide developers with tool-supported mechanisms during a product line's domain and application engineering phases to investigate the safety of the product line's requirements, design and architecture. Additionally, this effort explores how safety analysis assets can be reused within a product line as well as identifies the limits on reuse of product-line safety analysis assets.

The running example that we use to demonstrate our approach is a pacemaker product line. Previous work has been done in state-based modeling of a pacemaker by Goseva-Popstojanova et al. (2003) and Douglass (1999) and as examples with the Rhapsody toolset from I-Logix. However, previous work in state-based modeling of pacemakers considered the pacemaker as a single system rather than as a product line of models, as is done here. Goseva-Popstojanova et al., like us, is concerned with safety properties and uses SFMECA to identify software faults. These faults are then injected into a UML-based architectural design model to identify and evaluate their effects. Their approach helps identify which architectural components add risk and merit additional development resources but, unlike the work reported here, does not verify safety properties.

3. Approach

The technique described here for safety analysis of software product lines using state-based modeling consists of five steps. We describe each of the steps briefly in this section and then more rigorously in Section 4, where they are applied to the pacemaker product line.

3.1. Method overview

This section describes the five steps of our technique. Steps 1 through 4 are done in the domain engineering phase with the entire product line being taken into consideration. Step 5 is done in the application engineering phase with the verification being performed for each product member.

3.1.1. Commonality and variability analysis

The Commonality and Variability Analysis (CVA) (Weiss and Lai, 1999) is an established technique for providing domain definitions for the product line. It identifies the requirements for the entire product line (commonalities) and for specific product members (variabilities, or variations).

3.1.2. Hazard analysis

This step applies a hazard analysis technique called Software Fault Tree Analysis (SFTA) to the product line. A SFTA takes a hazard as the root node and identifies the contributing causes to it. A SFTA is a safety analysis technique widely used in high-assurance applications that assists domain engineers to systematically find causes of a certain hazard (an undesired event that can cause great loss) in a "top-down" manner. The nodes are hierarchically connected via AND or OR logic gates to describe the causal relationship to their parent nodes. The leaf nodes of the SFTA are basic events.

PL-SFTA is an extension of traditional SFTA to include the variations within a product line (Dehlinger and Lutz, 2004; Dehlinger and Lutz, 2006). It labels each leaf node of a SFTA with the commonality or variability associated with that leaf node. The commonality and variation information are taken from the Commonality and Variability Analysis (CVA) of the product line provided by the previous step of this technique.

A PL-SFTA considers all instantiations of the product line rather than a single system. To derive a particular product-line member's fault tree from the PL-SFTA, the PL-SFTA is pruned such that the resulting SFTA only contains those failures that are caused by the commonalities and the variations defining the specific product-line member. Interested readers are directed to Dehlinger and Lutz (2004) for details.

3.1.3. Variation model generation

In this step, the leaf nodes of the PL-SFTA are mapped into architectural components. The behaviors of each component are then modeled in a state chart model. This process starts from the product that has the fewest variations, meaning that most of its behaviors are shared by all the products in the product line. We then incrementally build the model by adding variations of other products and the associated dependencies as described in the next section. Such state models, once built, can be largely reused for

the safety analysis of other systems within the same product line.

3.1.4. Scenario derivation

We derive both required scenarios and forbidden scenarios from the PL-SFTA. Looking at the root-node of the PL-SFTA, if it (the hazard) or its negation (the safety property) can be mapped into a sequence diagram, we create state models to model the commonalities and variations of the components indicated in the leaf nodes of a current PL-SFTA. We call scenarios derived from the safety property required scenarios, and scenarios derived from hazards forbidden scenarios (Harel and Marelly, 2003).

If the root node cannot be mapped into testable scenarios (largely due to tool limitations, discussed in Section 5), we go to the root-node of the sub-tree and repeat the above process. By the end of the second step the original PL-SFTA should be fully covered. A PL-SFTA is fully covered if its root node, or each of its sub-tree's root nodes, is mapped into a sequence diagram, and its leaf nodes are captured in the state models.

3.1.5. Scenario-guided model analysis

In this step we take the safety-related scenarios and corresponding state model constructed above and fully exercise the scenarios against the state model. We say that a model is fully exercised if each of the legal combinations of commonalities, variations, and dependencies specified in the leaf nodes of the PL-SFTA is separately enabled in the state model and tested against the same scenario.

We use the TestConductor toolset (described below) to exercise the model. For required scenarios, if any test fails (the model execution does not match the specified scenario), the inconsistencies are identified in the state model and the design is updated. For forbidden scenarios, if the test shows that illegal behavior related to hazards is possible, we identify the cause in the state model and update the design.

The five steps described above are performed iteratively (i.e., if the output generated from a certain step affects previous steps, those steps need to be repeated) until no errors are found. For example, gaps in coverage found in Step 5 may result in specifying of additional product-line requirements in Step 1 or only to modeling errors. At that point, implementation or more formal verification is appropriate. Section 4 gives a detailed description of each of the steps.

3.2. Software tools

The integrated, visual development environment of Rhapsody is used to build the product-line statecharts. The process described above uses executable UML in the Rhapsody software modeling environment (Rhapsody, 2005) and the associated tool, TestConductor. Both are products from I-Logix.

The Rhapsody development environment supports the process activities of checking the model for inconsistencies, and of animating the sequence diagrams and the statecharts. The toolset also permits injection of inputs and events into the model during run time, and automated comparison of the designed sequence diagram with the animated sequence diagram to verify output events. Rhapsody is designed for real-time, embedded software development, making it well suited to the pacemaker product-line domain.

TestConductor provides a scenario-driven way to explore the behavior of the model as different variations are selected or de-selected, and as different values of variations are input. With TestConductor, multiple, distinct iterations through the statechart can be specified, with a new instance of an event or message being automatically generated each time. Thus, animation of multiple valid paths through a statechart can be executed, supporting checks that safety properties were satisfied. A limitation of the tool for the product-line application is that it does not handle time-out messages. We discuss in Section 5 how this affected the validation process.

3.3. Pacemaker product line

To illustrate the process outlined in Section 3.1, we use a pacemaker product line as a running example throughout Sections 4 and 5. A pacemaker is an embedded medical device designed to monitor and regulate the beating of the heart when it is not beating at a normal rate. A patient's need for a pacemaker typically arises from a slow heart rate (bradycardia) or from a defect in the electrical conduction system of the heart. A pacemaker consists of a monitoring device embedded in the chest area as well as a set of pacing leads (wires) from the monitoring device into the chambers of the heart (Ellenbogen and Wood, 2002). In our simplified example, the monitoring device has two basic parts: a sensing part and a stimulation part. The sensing part monitors the heart's natural electrical signals to detect irregular beats (arrhythmia). The stimulation part generates pulses to a specified chamber of the heart when commanded.

The timing cycle of our simplified pacemaker consists of two periods: a sensing period and a refractory period. Each pacemaker timing cycle begins with the sensing period, during which the sensor is on. If no heartbeat is sensed, a pulse will be generated at the end of the sensing period. The refractory period follows the sensing period but has the sensor off to prevent over-sensing (i.e., sensing the pulse it just generated). If a heartbeat is detected during the sensing period, no pulse is generated and the refractory period will be initiated. Thus, a timing cycle is the interval between two natural heartbeats, between two pulses, or between a heartbeat and a pulse, depending on the heart's behavior. The sensing period can vary between a lower rate limit and a higher rate limit, according to a patient's activity level.

Typically, pacemakers can operate in one of three modes: Inhibited, Triggered or Dual. Inhibited Mode is

when the sensed heartbeat inhibits stimulation and causes the pacemaker to restart the pacing cycle. Triggered Mode is when a sensed heart beat triggers stimulation. Dual Mode pacemakers have the ability to operate in either Inhibited or Triggered Mode.

In our example, we only consider a single-chambered product line of pacemakers that does pacing/sensing in the heart's ventricles. In actuality, some pacemakers are dual-chamber, and the pacing/sensing algorithms applied to each chamber can be different although highly coordinated. This paper considers three different products within the pacemaker product line: BasePacemaker, RateResponsivePacemaker and ModeTransitivePacemaker.

A RateResponsivePacemaker has additional sensors for detecting the patient's motion, breathing, etc. This allows the rate of the pacemaker to be responsive to the patient's current activity level. For example, when the patient is exercising, his or her heart rate will naturally be higher. A ModeTransitivePacemaker is a Dual Mode pacemaker that can switch between Inhibited Mode and Triggered Mode during runtime.

A safety property, motivates and explains the method: *the pacemaker shall always give a pulse to the heart when no heartbeat is detected during the sensing period*. The rationale behind this safety property is that when the heart has bradycardia symptoms (slow heart rate), the lack of heartbeat for a certain period is life threatening and thus must be treated with an electrical pulse. This safety property must hold for all systems in the pacemaker product line.

4. Product line safety analysis using state-based modeling

This section describes each of the five steps outlined in Section 3.1 in more detail and applies them to the pacemaker product-line example.

4.1. Commonality and variability analysis

Requirements and features for a product line are often specified in a Commonality and Variability Analysis (CVA) (Ardis and Weiss, 1997; Weiss and Lai, 1999). A CVA provides a comprehensive specification of the product line that details the shared, core requirements for all the products in the product line (i.e., the commonalities) and the requirements specific to only some products (i.e., the variations). This specification helps in providing pertinent domain definitions, the core set of product features and the scope of the product line. A portion of the CVA for the pacemaker product line used here is given in Fig. 1. The variations distributed among product-line members are shown in Table 1.

4.2. Hazard analysis

The hazard analysis uses Product-Line Software Fault Tree Analysis (PL-SFTA) both as a guide to deriving scenarios against which to test the models and to appropriately scope the level of detail needed in the models for safety analysis.

Commonalities

C1. A pacemaker shall have the following basic components: Controller, Sensor and PulseGenerator.
C2. A pacemaker shall be able to operate in the Inhibited Mode.
C3. A pacemaker's pacing cycle length shall be the addition of senseTime and refractoryTime.
C4. A pacemaker shall be able to set the senseTime to the LRL_rate of 800 msec.
C5. A pacemaker shall keep the refractoryTime set at 20 msec.
C6. A pacemaker shall be a single-chamber pacemaker.

Variations

V1. The senseTime of a pacemaker's pacing cycle may vary by setting the senseTime from LRL_rate of 800 msec to the URL_rate of 300 msec during runtime. [TRUE, FALSE]
V2. A pacemaker may transition from Inhibited Mode to Triggered Mode during runtime. [TRUE, FALSE]
V3. A pacemaker may have extra sensors to monitor a patient's motion, breathing, etc. [TRUE, FALSE]
V4. A pacemaker operating in Triggered Mode should confirm that a pulse is issued every time a heartbeat is detected. [TRUE, FALSE]
V5. A pacemaker operating in Triggered Mode should only use the LRL_rate of 800 msec as the senseTime. [TRUE, FALSE]

Dependencies

D1. A modeTransitive pacemaker must always confirm that a pulse is issued every time a heartbeat is detected while it is in Triggered Mode.
D2. A rateResponsive pacemaker must have additional sensors.
D3. A modeTransitive pacemaker must only use the LRL_rate setting for senseTime when it is operating in Triggered Mode.
D4. In a modeTransitive pacemaker, the rateResponsive function is valid only when the modeTransitive value is in Inhibited Mode.

Fig. 1. Excerpts from pacemaker product-line commonality and variability analysis.

Table 1
Products and their variations

Product name	Variations
BasePacemaker	
ModeTransitivePacemaker	V2, V4, V5
RateResponsivePacemaker	V1, V3
PL_Pacemaker	V1, V2, V3, V4, V5

4.2.1. Construct SFTA

The first activity is to construct the SFTA from the system requirements and design. A SFTA is a widely used backward safety analysis technique designed to trace the causal events of a specified hazard down to the basic faults of a single system (Leveson, 1995). The root nodes of fault trees are often the negation of a safety requirement. Root nodes may also be identified from preexisting hazard lists or from events with catastrophic effects in a Software Failure Modes, Effects and Criticality Analysis (SFMECA). In the case that an SFMECA exists, the generation of the SFTA can be partially automated (Dehlinger and Lutz, 2006).

In the pacemaker product-line example introduced in Section 3.3, the root node of SFTA is a negation of the safety property (S1): *the pacemaker fails to generate a pulse when no heartbeat is detected during the sensing period.*

4.2.2. Extend SFTA to include product line variations

The second activity is to extend the SFTA to include the variations in the product line. A PL-SFTA is an extension of traditional SFTA to include the variations within a product line (Dehlinger and Lutz, 2004). It labels each leaf node of a SFTA with a commonality or variation when applicable. Each leaf node of the SFTA is checked to find whether it is associated with one or several variations. If the node can be affected by the choice of variations, this leaf node is developed further (into a sub-tree) with the variability and commonality information added from the commonality analysis.

The PL-SFTA in Fig. 2 shows an example in the bottom left node "No pulse generated by the end of 300 ms sensing time". Each leaf node within this fault tree refers to either one of the commonalities or variations described in Section

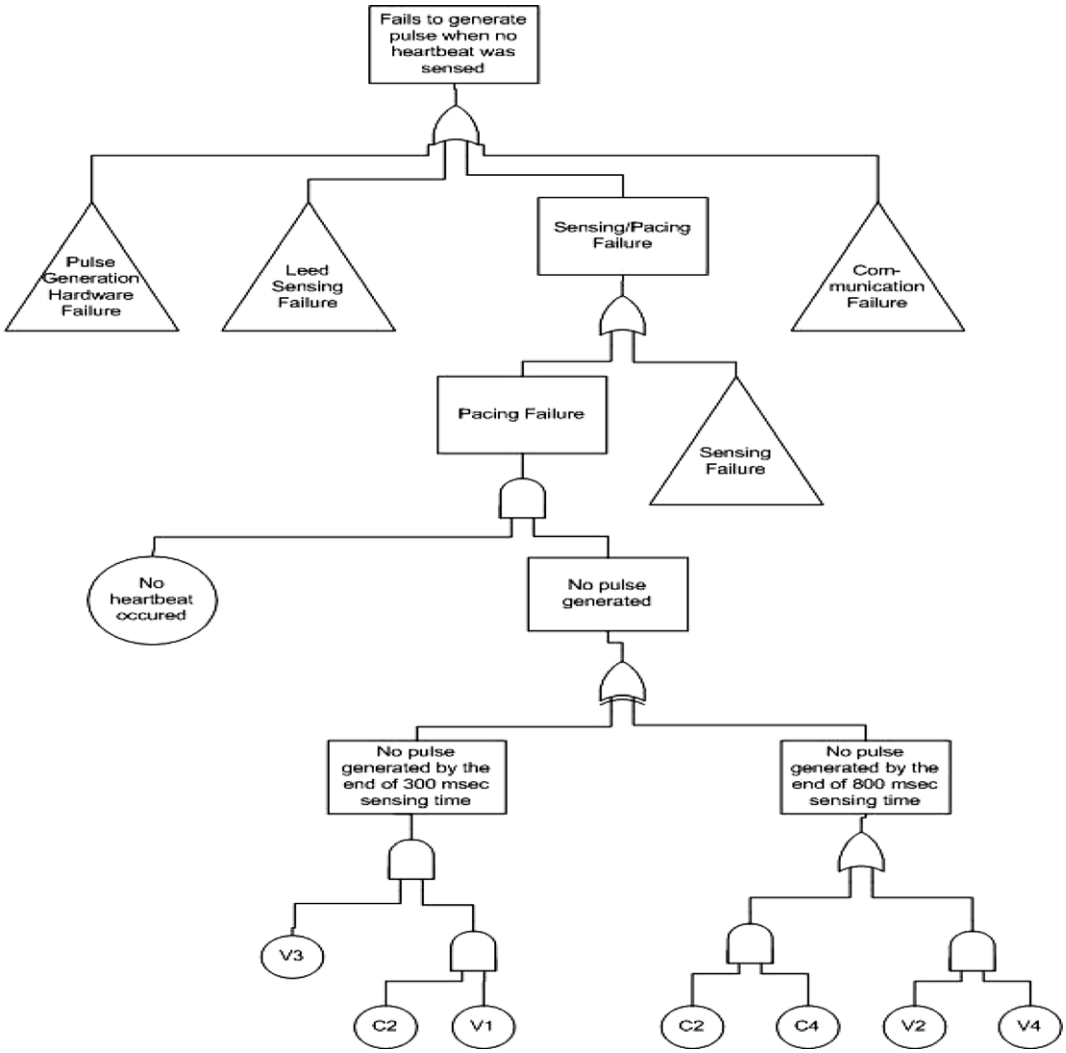

Fig. 2. Excerpt of pacemaker PL-SFTA.

3.3 to indicate which variations can contribute to the parent node's failure, or to a basic event, such as the node "No heartbeat occurred".

The preliminary safety analysis described above provides us with information regarding error-prone variation points from a product line point of view. This is necessary and helpful in that it introduces domain knowledge about variations into the analysis in a way that subsequent formal verification methods might find difficult to capture. Failure to adequately capture domain knowledge in safety analysis has been identified as a cause of accidents (Hanks et al., 2002).

However, the descriptive and static nature of the SFTA and SFMECA analysis makes it inadequate in terms of analyzing the dynamic feature interactions in a product line setting, and hence, in achieving asset reuse. By introducing state modeling and scenario-guided design analysis, such deficiencies can be addressed.

4.3. Variation model generation

The third activity is to map the leaf nodes of the PL-SFTA into components. The behavior of each component is then modeled in a state chart. Note that in this paper we assume the existence of a software architecture design, since the development of a product line architecture has been thoroughly addressed in e.g., Gomaa (2005) and Bosch (2000). Fig. 3 provides the UML Component Diagram that describes the software architecture for our product line. It consists of three major components: Pacemaker Controller, Detection, and Pulse Generator, each of which can be divided further into several sub-components. The different products in the product line share this architecture with the only difference being the presence or absence of some components (Liu et al., 2005a,b).

4.3.1. Associate leaf nodes with components

First, the corresponding component(s) of each of the leaf nodes in the PL-SFTA is obtained by looking at the system's architectural design. For example, the basic event "No heartbeat occurred" from Fig. 2 is generated by, and thus here associated with, the component "Heartbeat Simulator". Similarly, the leaf node V3 from Fig. 2 (that allows extra sensors) is tied to the component "ExtraSensor" in Fig. 3. Table 2 describes the mapping between the components in Fig. 3 and the leaf nodes in Fig. 2. (Note that C1 was not shown in Fig. 2 for readability.)

Table 2
The Mapping between leaf nodes and components

Leaf node	Component
Heartbeat occurred	Heartbeat simulator
V3	Extra sensor, motion simulator
V1, C2, C4, V2, V4	Pacemaker Controller
C1	Base sensor, pulse generator, Pacemaker Controller

4.3.2. Incrementally construct the variation model

Each component identified above is then modeled using state charts. The state model is built in an incremental fashion in order to model variations for different products in one state model. For example, we first model the Pacemaker Controller of the BasePacemaker (Fig. 4), and then incrementally add variations for the RateResponsivePacemaker and the ModeTransitivePacemaker products.

We briefly describe the process of incrementally constructing the product-line state model from a safety analysis perspective here.

- Creating BasePacemaker functionalities. Since every model in the product line shares the BasePacemaker functions, it is the baseline model. The BasePacemaker's behavior, shown in Fig. 4, has two states "On" and "Off". "On" is a composite (nested) state with two sub-states "Sensing" and "Refractoring". The pacemaker senses the heart beat in the "Sensing" sub-state and waits for the heartbeat or pace to complete in the

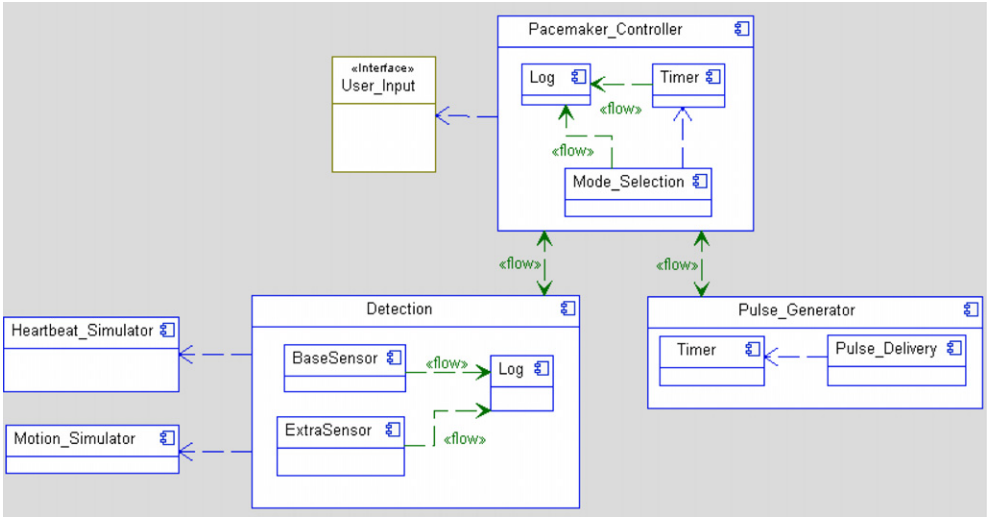

Fig. 3. Pacemaker architectural configuration.

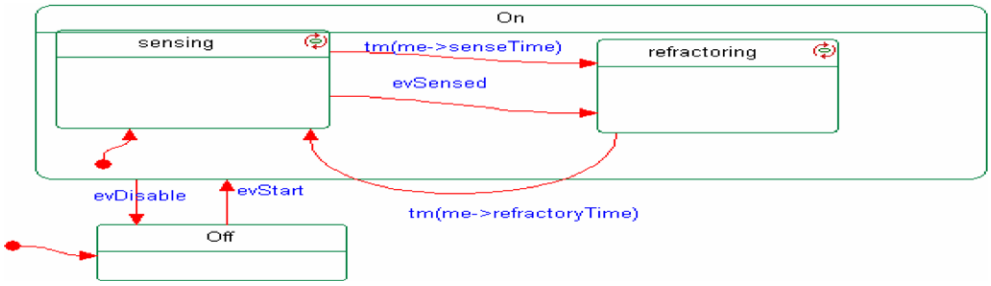

Fig. 4. Pacemaker controller in BasePacemaker.

"Refractoring" sub-state. This statechart displays the behavior that is common to all the pacemaker products in the product line.

- Adding RateResponsivePacemaker functionalities. The RateResponsivePacemaker's statechart inherits that of the BasePacemaker's. The variations V1 and V3 of the RateResponsivePacemaker are introduced into the model by adding orthogonal substates to existing states in the BasePacemaker's statechart. For example, V3 indicates that the RateResponsivePacemaker has additional sensors for detecting the patient's motion, breathing, etc, to allow the rate of the pacemaker to respond to the patient's current activity level. Thus, the "On" state in the statechart in Fig. 4 is expanded into an orthogonal state composed of two composite states: one with sub-states "Sensing" and "Refractoring" and the other with sub-states "LRL_rate" and "URL_rate".
- Adding ModeTransitivePacemaker functionalities. The statechart of the PL_Pacemaker inherits that of the RateResponsivePacemaker's, and adds functionalities from the ModeTransitivePacemaker. This is done by adding orthogonal substates and guarded transitions to existing states. For example, the variations V2 and V4 of the ModeTransitivePacemaker are introduced in the following manner: the "On" state in the statechart in Fig. 4 is expanded into an orthogonal state composed of three composite states: one with sub-states "Sensing" and "Refractoring", one with sub-states "LRL_rate" and "URL_rate", and the third one with sub-states "Inhibited_Mode" and "Triggered_Mode". The transitions between the "Sensing" and "Refractoring" sub-states have condition connectors showing that the behavior of these transitions are influenced by the choice of "Inhibited_Mode" and "Triggered_Mode". For example, when in "Triggered_Mode", the pacemaker stays in the "Sensing" sub-state until the sensed event (evSensed) is detected, at which point the pacemaker goes to the "Refractoring" sub-state. The condition connectors add guards to the transitions so that depending on the current value of the condition, the statechart takes different transitions and thus goes to different states. This incremental construction forms the state model for the PL_Pacemaker as it accommodates the variations of the three products in the product line.

The process of incrementally building up the product line variation model combines the Parameterized State Machines and Inherited State Machines methods described in Gomaa (2005). When a new variation is introduced, its corresponding statechart inherits the existing statechart and uses condition connectors (whenever necessary) to model dependencies between different variations.

Since our goal is safety analysis, the statechart for each component only models the behavior addressed by the leaf nodes of the PL-SFTA and any additional behavior relevant to the behavior. For example, the behavior of switching between the lower and upper rates for sensing a heartbeat, a variation of the RateResponsivePacemaker, is modeled in the statechart for Pacemaker Controller because it is associated with leaf node V1. However, the Log behavior in the Detection component, is not modeled because it is not associated with any leaf node.

When multiple leaf nodes correspond to a single component, the state chart for that component has to model the variations. These variations may not necessarily all reside in one product. For example, among the leaf-nodes that are associated with component "Pacemaker Controller", C2 and C4 belong to all three products in the product line, V1 and V3 belong to the RateResponsivePacemaker, and V2 and V4 belong to the ModeTransitivePacemaker.

The dependencies among variations that need to be modeled for the safety analysis were those where one variation's existence, value, or range of values depended on another variation's existence, value or range of values. For example, in the PL_Pacemaker, the RateResponsive function is valid only when the selected ModeTransitive variation is Inhibited Mode. If Triggered Mode or Non-Sensing Mode is selected, the RateResponsive variation is invalid. The general solution is to add states representing the change-of-value of the influenced variation, and to add guards on the transitions between those value-change-states in the influenced variation's statechart.

4.3.3. Represent binding time for variations

Another complicating factor is that in some real-world product lines, such as the pacemaker, a single variation may be able to be bound at different times. For example, the pacemaker's cycle length value can be bound at product architecture derivation time (in which case it is a BasePace-

maker), or – if it is a Rate-Responsive pacemaker – at either linking time (by the doctor) or at runtime (by an extra sensor). Similarly, some pacemakers have a programmable option that allows the pacing mode to be set at linking time, but changed at runtime through the mode transition function.

We thus found it necessary to model four possible binding times for variations according to the criteria in Svahnberg et al. (2005):

(1) *Product architecture derivation-time binding*. In our method, this is done in the state model generation step.
(2) *Compilation-time binding*. In our method, the code is automatically generated from the statechart model so this binding refers to whether or not to include a certain piece in the code-generation. For example, the Product-Line statechart models the behavior of both the BasePacemaker and the RateResponsivePacemaker. However, the system models a RateResponsivePacemaker only if ExtraSensor is selected in the code-generation; otherwise it models a BasePacemaker.
(3) *Linking-time binding*. This binding can be viewed as the initialization stage for operational execution. In our method, this is modeled by selecting the Rhapsody tool's "set parameter event" right at the start of animation. For example, the initial value of the cycle length parameter can be set when RateResponsivePacemaker is animated. This is described further in the next point.
(4) *Run-time binding*. An important application of our method is run-time variation checking, since other static analysis tools relating to product line variations, such as Decimal (Padmanabhan and Lutz, 2005) or PLFaultCAT (Dehlinger and Lutz, 2006), are not able to do run-time checking. In our method, this is realized by the injection of different events during animation. This can be done manually or through a simulator. Svahnberg, van Gurp and Bosch have described this additional aspect of binding. They define "internal binding" as occurring when the software contains the functionality to bind to a particular variant. As an example of internal binding, the UML condition connector proved to be a useful way to capture the variable behavior of the system in response to run-time changes in the values of variations, such as run-time switches between the different pacing modes. "External binding," on the other hand, occurs when there is a person (such as a doctor) or tool that binds the variation (Svahnberg et al., 2005). As an example of external binding, we can model the change in cycle length in a RateResponsivePacemaker by injecting the events evLRL_rate and evURL_rate from the ExtraSensor at run time.

4.4. Scenario derivation

The fourth activity is the derivation of forbidden scenarios from fault tree nodes and of required scenarios from the negation of fault tree nodes. Fault trees describe ways to push a system model to fail, or at least to find the vulnerable points in the system by indicating potential fault paths. The procedure to derive a scenario from a fault tree node follows.

4.4.1. Derive the initial scenarios, starting from the root node

Beginning at the root node of the fault tree, consider each lower level node. An intermediate node in the FTA is either a hazardous event or an event leading to a hazard. Given a node in the PL-SFTA, the sub-tree of such a node is initially treated as a black-box system with the input (stimuli from outside the black-box system) and output (response to the input by the black-box system) information extracted from the event description of the node. The input and output information are then depicted as a sequence diagram involving the black-box system and its environment in a sequence diagram. If input or output information cannot be extracted from a node, then the refinement of this node (its children nodes) is inspected to retrieve the information and derive scenarios.

For example, Fig. 5 models an initial forbidden scenario for the root node of the fault tree shown in Fig. 2, "Excerpt of Pacemaker PL-SFTA", as a sequence diagram. The root node event describes the hazard: no heartbeat was sensed during senseTime and no pulse was generated. In this case, there is neither input from the environment nor output from the system.

The scenario in Fig. 5 has two participants: the environment and the current system. The Tm(senseTime) denotes the timeout event of senseTime. Here senseTime is a general variable name that can be mapped to concrete values in a specific product-line member.

4.4.2. Refine the scenario to be testable

The above sequence diagram cannot be tested using the TestConductor tool because not all features of the sequence diagrams in message sequence chart (MSC) syntax are supported by the tool. In this case, the timeout

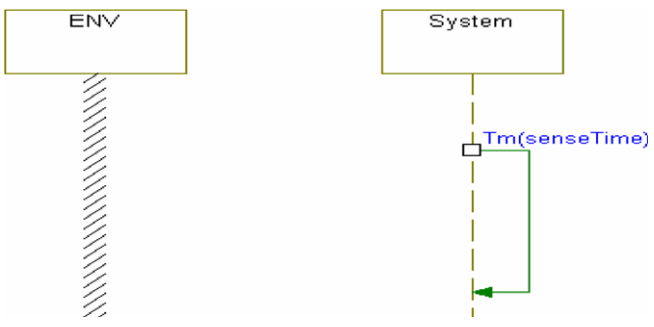

Fig. 5. Initially derived scenario.

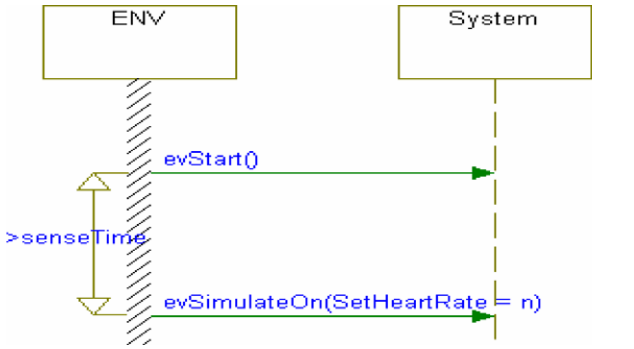

Fig. 6. Refined scenario.

event cannot be tested. Therefore, the above scenario is refined in order to make it testable by the TestConductor tool. The refined scenario is shown in Fig. 6.

The sequence diagram in Fig. 6 still has two participants: the environment and the current system. The time interval in the environment is used to replace the timeout event. This reflects the fact that, in implementing the system, a heart simulator will be needed to simulate heart beat signals. By setting the time interval between the time the system starts (invoked by the "evStart" event) and the time the heart simulator starts (invoked by the "evSimulatoOn" event) to be greater than the senseTime is then done, Fig. 6 depicts a scenario in which no heartbeat happens during sense time. No output from the system in the diagram indicates that no pulse is generated.

However, the above scenario is not generic enough in that it restricts the interval to begin at the time system starts. A more general scenario is to mark the start and end of the interval by evSimulateOff and evSimulateOn events, so that the pacemaker can start sensing at any time during the system execution. It is desirable to have the specified scenario as general as possible so the test results are not confined to a specific phase of system execution.

If no testable scenarios can be generated for a certain node, the analysis goes down one level in the PL-SFTA to derive scenarios from the children nodes if possible. In each case both the forbidden scenario and its negation (the required scenario) are inspected.

The goal of this step is to map either the root node of the PL-SFTA or all its children nodes into testable scenarios. The exit criteria for this step is that the test scenarios are at the same level of detail as the state model created in the last step (shown in Section 4.3) and that the PL-SFTA is fully covered. A PL-SFTA is fully covered if its root node, or each of its sub-tree's root nodes, is mapped into a sequence diagram, and the behaviors indicated by the leaf nodes are modeled by state models of their associated architectural components.

4.5. Scenario-guided model analysis

The fifth activity is to exercise the state model against the scenarios using the TestConductor tool to ensure that the safety properties previously identified in the hazard analysis always hold. TestConductor allows users to execute the state model, injecting messages on behalf of the environment or certain system components and monitoring the specified message sequences during execution time. It warns the user if any inconsistencies between the specified (or expected) scenario and the actual run-time scenario are captured.

4.5.1. Construct product line scenarios

Each scenario is verified against the state models, with one legal configuration of commonalities and variations enabled on the state models at a time. "Legal" here means that there is no violation of known product line dependencies (Padmanabhan and Lutz, 2005). In order to enable reuse of the sequence diagrams among the product-line members, we first specify a generic sequence diagram for the product line and then customize it according to the different configurations of variations. Note that although the state model was developed from a product line perspective, the model analysis must be done member by member in the product line. Therefore all the generic variables/message names in the sequence diagrams have to map to concrete values during testing.

Fig. 7 shows the sequence diagram for an example scenario derived from the root node hazard, "Fails to generate pulse when no heartbeat was sensed". This is a generic test scenario for this product line. A generic scenario includes all the components whose state models have been created in the second step. Each component is represented as a separate instance line. This includes the components shared by all the products (e.g., BaseSensor, PulseGenerator, etc.), components that have variation models (e.g., PL_Pacemaker), components that are variations themselves (e.g., ExtraSensor), and black-box components that generate the required environmental input in a real-time fashion (e.g., HeartSimulator and MotionSimulator).

The generic sequence diagram must also include the external events representing messages generated from the system border of the sequence diagram, e.g. the evStart( ) event in Fig. 7, and internal events that occur between the internal instances of the sequence diagram, e.g., the evPulseGeneratorOn( ) event in Fig. 7. Variations of data associated with the events, if possible, are also specified here. For example, in Fig. 7, "n" in the event "evSimulateOn (SetHeartRate = *n*)" is a parameter that shows different heart rate.

The generic test sequence diagram is then customized into different cases by adding variations until all the combinations of commonalties and variations indicated in the leaf nodes of the PL-SFTA are covered. This customization is most readily done hand-in-hand with the state model customization so that if a certain variation is represented in the sequence diagram, it is enabled in the state model as well.

4.5.2. Verify the state model

Each customized state model is now executed against its corresponding scenarios. In this case study, we executed

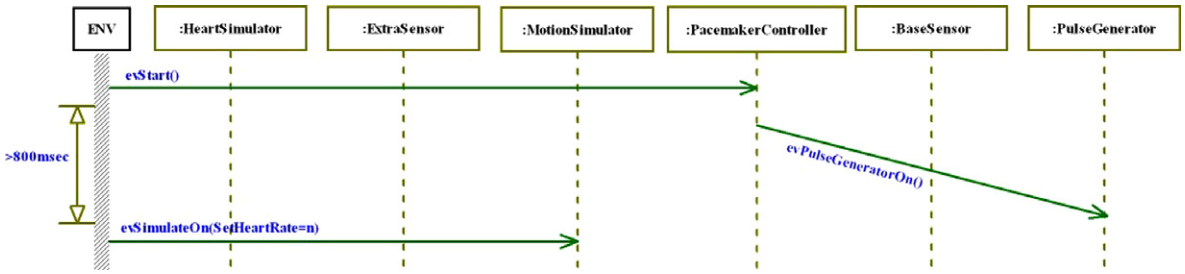

Fig. 7. Scenario derived from PL-SFTA.

the state model of PL_Pacemaker with no variation, with "InhibitedMode" and "TriggeredMode" enabled separately, and with the combination of "InhibitedMode" and "RateResponsiveMode" enabled. For each configuration, if the state models' execution traversed the events in the right order as specified in their customized scenario, then the configuration passed the test. These tests were run for all the product-line members BasePacemaker, RateResponsivePacemaker, ModeTransitivePacemaker (with InhibitedMode and Triggered Mode enabled separately), and for the PL_Pacemaker (with InhibitedMode and RateResponsive features combined).

The TestConductor showed that all the products passed their tests. The criterion for passing each test was that the execution of the model traversed the events specified in the named scenario in the right order. Fig. 8 is an example of the test results from validating the safety property S1 with Inhibited Mode and RateResponsive enabled. The safety property S1, the pacemaker fails to generate a pulse when no heartbeat is detected during the sensing period, was validated in all the tests we ran on the model.

Besides using TestConductor, potentially hazardous scenario can be confirmed and further revealed by monitoring the execution sequence generated during model animation, and comparing it with the scenarios derived from the fault tree. For example, the scenario that included an extra sensor (V3) revealed a single-point failure that had to be corrected as described in Section 5.1.2.

5. Discussion of results

This section briefly discusses the benefits and limitations of the approach described in this paper.

5.1. Applying the results to enhance safety

There are three main uses of the results to enhance the safety of the product line in this application: finding design errors, addressing safety-related concerns and scoping models for formal verification.

5.1.1. Finding design errors

TestConductor identifies inconsistencies between the design sequence diagram and the sequence diagrams generated during state model execution, e.g., messages out of order, wrong messages or missing messages. Such inconsistencies can readily be traced back to the state model that generates the message to determine the cause. For both required scenarios and forbidden scenarios, if the cause of the inconsistency is not related to incorrect modeling or limitation of the tool, the design should be updated to remove the identified problem.

To update the design to remove any identified problems, new product-line requirements (i.e., commonalities, variabilities and/or dependencies) may need to be included into the Commonality and Variability Analysis (CVA), described in Section 4.1. The updating of the CVA with

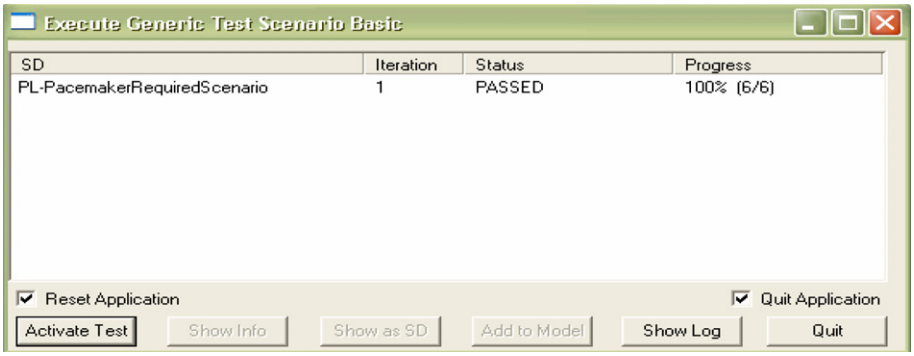

Fig. 8. Output from TestConductor.

new product-line requirements may then require additions/modifications to the product line's hazard analyses, described in Section 4.1 to account for the new possible hazards introduced by the new requirements. Following this, an additional iteration of Steps 3–5, described in Sections 4.3–4.5, would be necessary to assess the safety properties of the product line given the new product-line requirements.

5.1.2. Addressing safety-related concerns

One of the big benefits of the state-based modeling technique is that it checks if safety-related concerns rising from the hazard analysis are really credible threats, validates any mitigation steps and discovers discrepancies between static hazard analysis and system behavior. For example, from the PL-SFTA it was not found that the extra sensor (V3) was a potential single-point failure for the safety property S1. By disabling the extra sensor during the model execution, the scenario during model execution was compared with the required scenario and the following inconsistency was found: The upper scenario in Fig. 9 shows that the sensing interval is required to be less than 300 ms when V3 is present and the patient is at exercise. The lower scenario in Fig. 9 shows the actual scenario from execution. In it, the sensing interval is still 800 ms due to failed extra sensor. This demonstrates that V3 is indeed a credible potential vulnerability that can lead to a hazard, i.e., the failure to provide a pulse when the pacemaker should generate one.

Through model execution with different combinations of variations enabled, we investigated a possible mitigation, which was to add a new product-line safety requirement: if the pacemaker is currently working in Triggered or Inhibited Mode and the sensor fails, the pacemaker should automatically transition to Non-Sensing Mode. Since in the Non-Sensing Mode a continuous pulse can be generated automatically at least until the sensor recovers, this avoids the single-point failure.

In general, by adding a new product-line safety requirement, a commonality, variability and/or dependency could be introduced into the product line's CVA. This, again, may require an addition/modification to the hazard analyses and an additional iteration through Steps 2–5, as described in Sections 4.2–4.5 to validate the product line's safety properties with the new requirements.

5.2. Limitations

Although Rhapsody's executable state models support real-time notions, we found that it cannot enforce exact real-time measurement as required in some of the safety properties in our pacemaker case study. Therefore, the state-based modeling technique described in this work is not suitable for testing border time values as is often required in safety-critical, real-time systems. Rather, using our technique and Rhapsody, validating/testing the ordering logic and relative timing of failure events is possible.

Due to the limitations of the scenario description language used by the testing tool and to limitations of the testing tool itself, some scenarios were not "testable". For example, the timeout message (as shown in Fig. 5) and canceled-time out message cannot be tested by the TestConductor tool. Time intervals have to be between two messages sent from the system border to be testable. Also, Rhapsody supports Live Sequence Charts (Harel and Marelly, 2003) while TestConductor only supports UML sequence diagrams. By representing these non-testable scenarios in alternative ways (as in Section 4.4.2), some generality is lost in depicting the testing scenario. This seems to

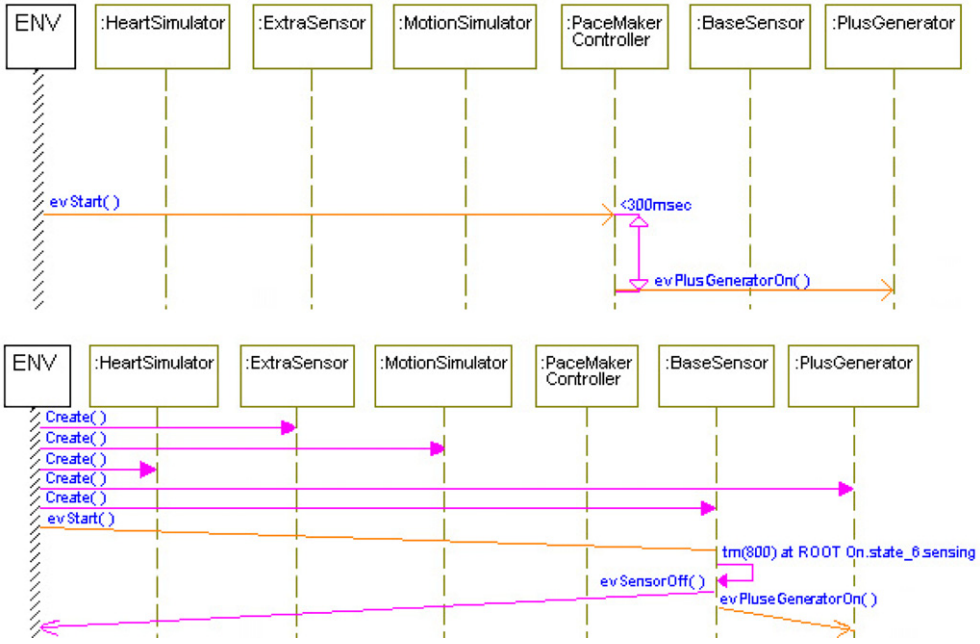

Fig. 9. Comparison between the required scenario and the actual scenario (when V3 is present).

be an unavoidable problem at the moment but future tool development may ease this difficulty.

5.3. Future directions

The technique described in this paper has only been applied to a simplified product line to date. Work is under way to investigate this approach on a large, real-world product line. We believe that it can be extended to analyze larger product lines because it uses PL-SFTA as a preliminary step. PL-SFTA decomposes the problem domain in a hierarchical way to control variations and commonalities related to certain safety properties. Modeling the variations in the state model also allows us to analyze relatively complex interactions within large product lines. In addition, the creation of test scenarios and state models are done in a reusable fashion to reduce the workload as the number of system grows.

The results of our analysis provided useful information for further verification as well. Exercising the state models made evident which parts relate to a certain safety property. The formal models can be derived from the state models relevant to that safety property. The scope needed for formal verification thus can be reduced. Similarly, the temporal logic properties are made more visible through the forbidden scenario and required scenario derivation. Problems found in formal verification can also be traced back to concrete scenarios in order to update the design using scenario execution as described earlier. In future work we plan to investigate how integrating the product line state-based modeling approach with formal verification can reduce the workload of formal verification.

6. Conclusion

The work described here provides guidelines for constructing the behavioral model of a product line's significant, safety-related variations in order to support automated verification of safety properties across the product line. Briefly, our method checks whether the variations and the behavior they introduce jeopardize the safety properties. The contributions of the paper are to show: (1) how to build a state-based, product-line model that can accommodate different types of variations and (2) how to extend scenario-guided execution of a model to verify product-line safety properties. By making it more practical to check variable behaviors for safety consequences, this method can enhance reuse in high-integrity product lines. In addition, by helping to manage the complexity introduced by variations, the method supports the potential reuse of previously performed safety analyses as new products are added to the product line.

Acknowledgements

This research was supported by the National Science Foundation under grants 0204139, 0205588, and 0541163. We thank Jeffrey Thompson, Guidant Corporation, for insightful feedback on preliminary versions of our approach and I-Logix for the use of their tools. The first author thanks Samik Basu for helpful discussions.

References

Ardis, M.A., Weiss, D.M., 1997. Defining families: the commonality analysis. In: Proceedings of the 1997 (19th) International Conference on Software Engineering, 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.

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

Campbell, L., Cheng, B.H.C., McUmber, W.E., Stirewalt, R.E.K., 2002. Automatically detecting and visualizing errors in UML diagrams. Requirements Engineering Journal 7 (4), 264–287.

Clauss, M., 2001. Modeling variability with UML. In: Proc. of Net.ObjectDays 2001, Young Researchers Workshop on Generative and Component-Based Software Eng., Erfurt, Germany, September 2001, pp. 226–230.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addision-Wesley.

Czerny, B.J., Heimdahl, M., 1998. Automated integrative analysis of state-based requirements. In: Proc. 13th IEEE Int'l conf. Automated Software Eng. (ASE'98), Honolulu, HI, pp. 125–134.

Dehlinger, J., Lutz, R.R., 2006. PLFaultCAT: a product-line software fault tree analysis tool. The Automated Software Engineering Journal 13 (1), 169–193.

Dehlinger, J., Lutz, R.R., 2004. Software fault tree analysis for product lines. In: Proc. 8th Int'l Symp., High Assurance Systems Eng. (HASE '04), Tampa, FL, pp. 12–21.

Dehlinger, J., Lutz, R.R., 2005. A Product-line approach to safe reuse in multi-agent systems. In: Proc. 4th Int'l Workshop on Software Eng. For Large-Scale Multi-Agent Systems (SELMAS'05), St. Louis, MO, pp. 83–89.

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. Addison-Wesley, MA.

Ellenbogen, K.A., Wood, M.A., 2002. Cardiac Pacing and ICDs. Blackwell Science, Inc.

Feng, Q., Lutz, R.R., 2005. Bi-directional safety analysis of product lines. Journal of Systems and Software 78 (2), 111–127.

Gomaa, H., 2005. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley.

Goseva-Popstojanova, K., Hassan, A., Guedem, A., Abdelmoez, W., Nassar, D.E.M., Ammar, H., Mili, A., 2003. Architectural-level risk analysis using UML. IEEE Transactions on Software Engineering 29 (10), 946–960.

Hanks, K.S., Knight, J.C., Holloway, C.M., 2002. The role of natural language in accident investigation and reporting guidelines. Workshop on the Investigation and Reporting of Incidents and Accidents, Glasgow, Scotland.

Harel, D., Marelly, R., 2003. Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer.

Kang, K.C., 2006. Software product line research topics. In: Proc. 10th Int'l Softwarae Product Line Conf, pp. 103–112.

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 5, 143–168.

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

Liu, J., Lutz, R., Thompson, J., 2005a. Mapping concern space to software architecture: a connector-based approach. In: ICSE 2005

Workshop on Modeling and Analysis of Concerns in Software (MACS'05), St. Louis, MO, pp. 1–5.

Liu, J., Dehlinger, J., Lutz, R., 2005b. Safety analysis of software product lines using state-based modeling. In: 16th IEEE International Symposium on Software Reliability Engineering, pp. 21–30.

Liu, J., Dehlinger, J., Sun, H., Lutz, R., in press. In: Proc. 5th Workshop on Model-Based Development for Computer-Based Systems, Tucson, AZ.

Lutz, R.R., 2000a. Extending the product family approach to support safe reuse. Journal of Systems and Software 53 (3), 207–217.

Lutz, R.R., 2000b. Software engineering for safety: a roadmap. In: Finkelstein, A. (Ed.), The Future of Software Engineering. ACM Press.

Lutz, R.R., Woodhouse, R.M., 1999. Bi-directional analysis for certification of safety-critical software. In: Proc. 1st International Software Assurance Certification Conference (ISACC'99), Washington, DC.

Mellor, S.J., Balcer, M.J., 2002. Executable UML: A Foundation for Model Driven Architecture. Addison-Wesley.

Padmanabhan, P., Lutz, R.R., 2005. Tool-supported verification of product line requirements. Automated Software Engineering 12 (4), 447–465.

Rhapsody Tutorial for Rhapsody in C., 2005. Release 6.0 MR-1. I-Logix, Inc.

Schwanke, R., Lutz, R.R., 2004. Experience with the architectural design of a modest product family. Software Practice and Experience 34 (13), 1273–1276.

Svahnberg, M., van Gurp, J., Bosch, J., 2005. A taxonomy of variability realization techniques: research articles. Software – Practice and Experience 35 (8), 705–754.

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

Jing (Janet) Liu is currently a Ph.D. student in the Department of Computer Science at Iowa State University, Ames, IA. She received a B.E. in Computer Science and Technology from East China University of Science and Technology. Her research interests include software product lines, software safety analysis, formal verification and testing, model-based development, and aspect-oriented software development. Her research is supported by the National Science Foundation.

Josh Dehlinger is currently a Ph.D. student in the Department of Computer Science at Iowa State University, Ames, IA. He received a B.S. in Management of Computer Systems from the University of Wisconsin-Whitewater. His research interests include software product-line engineering, software safety analysis and software engineering for multi-agent systems. His research is supported by the National Science Foundation.

Robyn R. Lutz is a 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.