Safety Analysis of Software Product Lines Using State-Based Modeling

Jing Liu
Computer Science Department
Iowa State University
janetlj@cs.iastate.edu

Josh Dehlinger
Computer Science Department
Iowa State University
dehlinge@cs.iastate.edu

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

Abstract

The analysis and management of variations (such as optional features) are central to the development of safety-critical, software product lines. However, the difficulty of managing variations, and the potential interactions among them, across an entire product line currently hinders safety analysis in such systems. The work described here contributes to a solution by integrating safety analysis of a product line with model-based development. This approach provides a structured way to construct a state-based model of a product line having significant, safety-related variations. The process described here uses and extends previous work on product-line Software Fault Tree Analysis to explore hazard-prone variation points. The process then uses scenario-guided executions to exercise the state model over the variations as a means of validating the product-line safety properties. Using an available tool, relationships between behavioral variations and potentially hazardous states are systematically explored and mitigation steps are identified. The paper uses a product line of embedded medical devices to demonstrate and evaluate the process and results.

1. Introduction

The analysis and management of variations (such as optional features) are central to the development of safety-critical, software product lines. However, in safety-critical product lines such as pacemakers [11], mobile communication devices for emergency workers [9], constellations of satellites [6], and medical-imaging systems [24], two obstacles to the management and analysis of variations have hindered safety analysis. First, many safety-critical product lines involve real-time embedded software. Such software can make it difficult to provide assurance that safety properties will be satisfied in the presence of variations. This discourages the addition of variations and limits the potential for reuse within the product line. Second, a safety-critical product line must satisfy its safety properties in all allowable configurations (i.e., choices of variations).

Designers, engineers and developers currently lack a structured and efficient decision mechanism to determine whether a particular variation will safely integrate into the product line without introducing safety concerns or hazardous states. We define software variations as “the ability of a software system or artifact to be changed, customized, or configured for use in a particular context” [4]. Therefore, we introduce a technique to do safety analysis for product lines based on state-based modeling. This technique uses current safety analysis techniques (SFTA, SFMECA, described in Section 3.1) and, at the same time, accommodates product-line variations.

The technique described here performs safety analysis with the assistance of an executable statechart model and sequence diagrams. UML models are commonly developed during the design phase of initial product line development. By integrating existing UML models and state-based modeling tools that are already widely used in industry with hazard analysis techniques, we hope to encourage safer and more reliable software product lines. This approach can help identify and support reuse of safety-analysis assets within a product line. Similarly, by focusing specifically on the technical challenges involved in the safety analysis of systems with software variations we hope to encourage improved safety analysis of critical product line software.

This paper describes the results of an investigation into how to perform safety analysis on the variations in a product line using state-based modeling. The contribution of the paper is to make it more practical to check that safety properties for the product line are satisfied in the presence of variations through scenario-guided execution, or animation, of the model. Performing analysis at the design level allows safety engineers to discover faults early enough to design mitigation strategies before development and deployment. Relationships between the behavioral variations and hazardous states can be systematically explored. The improved management and analysis of variations promotes safer reuse of assets in product lines.

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

The rest of this paper is organized as follows. Section 2 presents needed background information and related work. Section 3 describes the approach and introduces the running example (a pacemaker product line). Section 4 addresses the state-based modeling of the product-line's commonalities, variations and dependencies using executable statecharts from a software-safety perspective. Section 5 describes the subsequent model-based validation of safety properties and identification of additional safety concerns, and evaluates the method on the pacemaker product line. Finally, Section 6 offers some concluding remarks.

2. Background & Related Work

Our work is based on the overlapping areas of model-based development, software product-line engineering 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.

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 development. State-based modeling has been shown to support verification of behavioral requirements [5]. 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 [2]. Executable UML allows animation of scenarios to test the adequacy of the models that have been built [22]. More recently Gomaa has shown how executable UML statecharts can be used for product-line models [13].

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 [4, 26]. The product-line engineering concept is advantageous in that it exploits the potential for reusability in the analysis and development of the core and variable requirements in each member of the product line.

While work to date concentrates on how to validate whether a single system's model behaves correctly, this work 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 consistent with several widely-used product-line engineering frameworks. A variety of feature diagrams or variation models, including those in Kobra [1], FORM [16] and FAST [26], can be annotated with references to the associated modeling elements in this work. Sophisticated tool support allows the animation of a state-based design model [10]. Scenario-guided testing of state-based design models has been previously shown to provide a powerful way to identify omissions in requirements [15].

There has been widespread attention to UML modeling of product lines, mainly from the perspective of promoting reusable components. Clauss extends UML to support feature diagrams and adds elements describing variations to the UML package diagram [3]. Doerr categorizes the relationships that exist in a variation model and ties these to UML concepts [9].

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 [17] 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 [21]. More recently, a bi-directional approach was extended to product lines [12]. Software Fault Tree Analysis [17] and Software Failure Modes, Effects and Criticality Analysis (SFMECA) were also adopted for use in product-line safety analysis [7, 8, 19].

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 [14], by Douglass [10] and by I-Logix [23]. 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, 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. This approach helps identify which architectural components add risk and merit additional development resources but does not verify safety properties, as done here.

3. Approach

The method described here for safety analysis of software product lines using state-based modeling consists of four steps. We describe the steps briefly here and then more fully in Section 4, where it is applied to the pacemaker product line.

3.1. Method Overview

This section describes the four steps of our technique.

1) ***Hazard Analysis.*** We apply each Product-Line Software Fault Tree Analysis (PL-SFTA) taking hazards as root nodes, to identify contributing causes to each root node. PL-SFTA is an extension of traditional SFTA to include the variations within a product line [7, 8]. A PL-SFTA considers all instantiations of the product line rather than a single system. To derive a product-line member's fault tree from the PL-SFTA, we prune the tree(s) such that the resulting SFTA only contains those failures that are caused by the commonalities and the variations defining the product-line member. Interested readers are directed to [7] and [8] for details.

2) ***Scenario Derivation.*** 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 cut-set of a current PL-SFTA. We call scenarios derived from the safety property *required scenarios*, and scenarios derived from hazards *forbidden scenarios* [15].

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 cut-set, or that of all of its sub-trees, are modeled as state models.

3) ***Variation Model Generation.*** The PL-SFTA not only points out the vulnerable variation points, but also the scope needed for the state modeling in order to perform the safety analysis. For each PL-SFTA (or sub-tree) whose root node (or negation of the root node) can be mapped into a testable scenario, the components involved in the cut-set are each modeled in the *minimum variation model* required for the safety analysis. A minimum variation model consists of only the commonalities and variations specified in the cut-set of that PL-SFTA (or sub-tree), and are modeled in one state model. The specific technique to model variations for different products in one state model is described below. By modeling the minimum variations, the complexity of each state model is significantly reduced and the scalability of the technique preserved. By modeling all the variations in a single state model, the interactions between variations can be easily tested, plus the modeling effort can be reused throughout the product line.

4) ***Scenario-Guided Model Analysis.*** In the 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 cut-set of the PL-SFTA is separately enabled in the state model and tested against the same scenario.

We use TestConductor (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 any test fails we check if the inconsistencies are related to the hazard identified in the scenario. If the test shows that illegal behavior is possible, we identify the cause in the state model and update the design.

The four steps described above are performed iteratively until no errors are found. At that point, implementation or more formal verification of performance is appropriate. We give a detailed description of each of the steps in Section 4.

3.2. Software Tools

The second step of the process described above uses executable UML in the Rhapsody software modeling environment [23]. The third step employs the associated tool, TestConductor. Both are products from I-Logix.

We used the integrated, visual development environment of Rhapsody to build the product-line statecharts. The Rhapsody development environment supported our process activities of checking the model for inconsistencies, 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 provided a scenario-driven way to explore the behavior of the model as different variations were selected or deselected, and as different values of variations were input. With TestConductor, multiple, distinct iterations through the statechart can be specified,

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 pacing mode to Triggered pacing 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 pacing mode should confirm that a pulse is issued every time a heartbeat is detected. [TRUE, FALSE]
V5. A pacemaker operating in Triggered pacing 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 pacing 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 pacing mode.
D4. In a modeTransitive pacemaker, the rateResponsive function is valid only when the modeTransitive value is in Inhibited mode.

Figure 1. Excerpts from pacemaker product-line Commonality and Variability Analysis

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 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 typically 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 [11]. 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 specified chamber of the heart when commanded [11]. 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. 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, two pulses, or a heartbeat and a pulse depending on the heart's behavior.

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 simplified example, we only consider a single-chambered product line of pacemakers that does pacing/sensing in the heart's ventricles. The Commonality and Variability Analysis (CA) is shown in Figure 1. We consider three different products within the pacemaker product line: BasePacemaker, RateResponsivePacemaker and ModeTransitivePacemaker. The product-line hierarchy is shown in Figure 2.

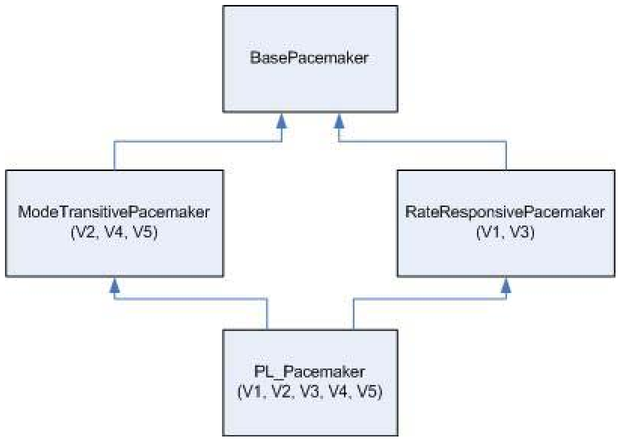
BasePacemaker
ModeTransitivePacemaker (V2, V4, V5)
RateResponsivePacemaker (V1, V3)
PL_Pacemaker (V1, V2, V3, V4, V5)

Figure 2. Pacemaker Product-Line Hierarchy

Here we use a safety property (S1) to motivate and explain our method: In the Inhibited pacing mode, the pacemaker should 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, the lack of heartbeat for a certain period is life threatening and thus should 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

In this section, we describe each of the four steps we outlined in Section 3.1 in more detail and apply them to the pacemaker product-line example.

4.1. Hazard Analysis

In this work, we use the PL-SFTA as a guide in deriving scenarios against which to test our model as well as to appropriately scope the model's needed level of detail for safety analysis. Using the PL-SFTA, we can derive forbidden scenarios from the paths leading to the root node of any give fault tree. It is with these forbidden scenarios we want to test our state-based model to ensure these situations cannot happen. We utilize the PL-SFTA to limit the scope of the state-based model by examining the remaining leaf nodes of the fault tree after pruning the PL-SFTA to include only those variations for the member currently under consideration. Examining the leaf nodes gives a heuristic for identifying which components of the product-line member must be within the state model to test for the unsafe conditions (the forbidden scenarios).

A representative PL-SFTA is shown in Figure 3. Each leaf node within this fault tree refers to one of the commonalities or variations described in Section 3.3 to indicate which variations can contribute to the parent node's failure.

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 into the analysis in a way that subsequent formal verifications find difficult to achieve. However, the descriptive and static nature of such analysis makes it inefficient in terms of in 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.

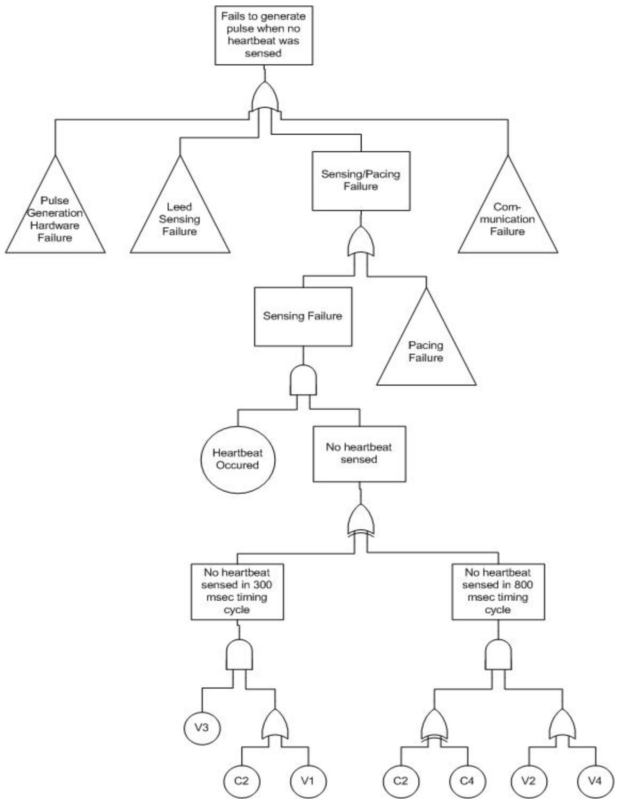
Fails to generate pulse when no heartbeat was sensed
Pulse Generation Hardware Failure
Leed Sensing Failure
Sensing/Pacing Failure
Com-munication Failure
Sensing Failure
Pacing Failure
Heartbeat Occured
No heartbeat sensed
No heartbeat sensed in 300 msec timing cycle
No heartbeat sensed in 800 msec timing cycle
V3
C2
V1
C2
C4
V2
V4

Figure 3. Excerpt of Pacemaker PL-SFTA

4.2. Testable Scenario Derivation

We derive testable scenarios from the PL-SFTA. Figure 4 shows an example scenario derived from the root node hazard, “Fails to generate pulse when no heartbeat was sensed”. This is a common test scenario for the product line. By “common test scenario”, we mean that every system in the product line must satisfy the safety property that negates this hazard. Thus, the components involved in the scenario include all components related to the safety property in the product-line scope.

Here, the event “evPulseGenerator” occurs between “evStart”(the event to start the PL_Pacemaker) and “evSimulateOn(SetHeartRate=n)” (the event to start the HeartSimulator to mimic the heart beat activities) in order to specify that when there is no heartbeat (i.e., before evSimulateOn), there should be a pulse-generated event (evPulseGenerateOn) detected. Note that the time interval “>800ms” specified between “evStart” and “evSimulateOn(SetHeartRate=n)” means that no heartbeat would occur during senseTime (we specified senseTime=800ms in the model).

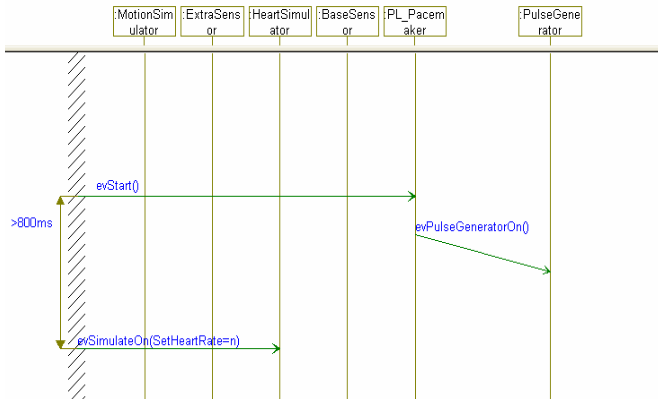

Figure 4. Scenario Derived from PL-SFTA

4.3. Variation Model Generation

To model variations for different products in one state model, the state model is built in an incremental fashion. We start from the product that has the fewest variations, which we call the “near-common-model” [20], meaning that most of its behaviors are shared by all the products in the product line. We then incrementally build the model with variations of other products and the associated dependencies as described in the subsequent section. Note that such state models, once built, can be largely reused for the safety analysis of other systems within the same product line. In addition, when new commonalities, variations or dependencies are introduced, the variation model can be readily updated to see if the safety properties still hold.

Note that components other than those that have been identified in the PL-SFTA cut-set but which have interactions with cut-set leaves are simply treated as black box. They are modeled in simple state models which generate only the needed message. The generation of those messages should not depend on the behaviors of the components included in the cut set. If they do, the PL-SFTA must be updated and the message events included in the cut set as well. In our pacemaker example, we modeled a heart simulator (to provide the heartbeats to the model) as one such black box component.

Although each of the state models is a simplified version (including the commonalities and variations related to the hazard of concern only), the modeled part should still match the design specification of that part. In other words, the correctness of the state model itself, with regard to the conformance of the functional requirements of the modeled part, should be assessed. This can be achieved by means of model debugging using functional scenarios, as described in [15].

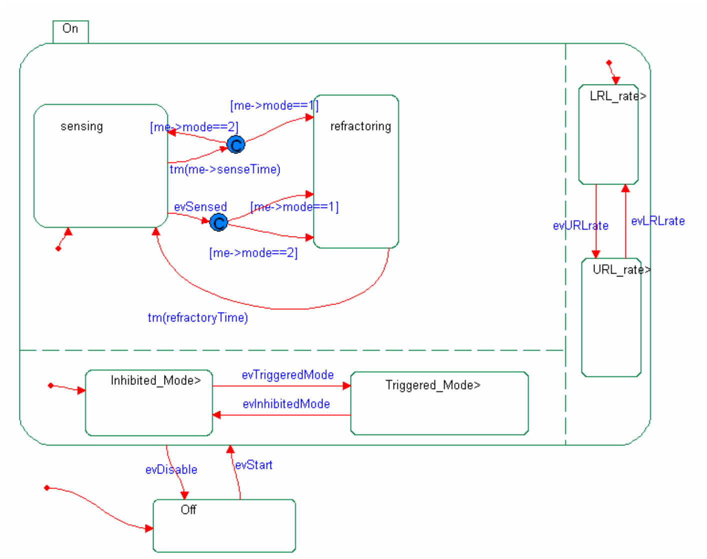

Figure 5. Product-Line statechart, PL_Pacemaker

In this case study, for the safety property S1, the BasePacemaker is such a “near-common-model”. Every model in the product line shares the BasePacemaker functions. Its behavior can be represented in a statechart (omitted here for space reasons), with two states “On” and “Off”. The “On” is a composite (nested) state with two sub-states “Sensing” and “Refractoring”. This statechart displays the behavior that is common to all the pacemaker products in the product line. Different products in the pacemaker product line require additional behavior beyond that of the BasePacemaker. For example, the 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 (e.g., at rest vs. exercising). Thus, the “On” state in its statechart model is an orthogonal state composed of two composite states: one with sub-states “Sensing” and “Refractoring”, one with sub-states “LRL_rate” and “URL_rate”. LRL is the lower rate limit with sense time of 800sec; URL is the upper rate limit with sense time of 300sec.

Similarly, the ModeTransitivePacemaker also inherits from BasePacemaker but has some additional states not present in either BasePacemaker or RateResponsivePacemaker. For example, its "On" state is an orthogonal state composed of two composite states: one with sub-states "Sensing" and "Refractoring", one with sub-states "Inhibited_Mode" and "Triggered_Mode".

In order to have all the variations specified in one state model, we construct the state model for PL_Pacemaker (see Figure 4). Its "On" state is a orthogonal state composed of three composite states: one with sub-states "Sensing" and "Refractoring", where the transitions between them have condition connectors showing that the transition is influenced by RateResponsive variation and ModeTransition Variation separately; one with sub-states "Off" and "On", where "On" is composed of sub-states "LRL_rate" and "URL_rate"; one with sub-states "Inhibited_Mode" and "Triggered_Mode".

We now describe the differences in modeling and validation techniques based on differences in binding time of the variations and on dependencies among the variations.

4.3.1. Modeling variations. A standard solution to the representation of variations is to use variation points. For example, some variations can be modeled as features that set a value or turn on/off during different stages of product line life cycle [11]. One 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 design time (in which case it is a BasePacemaker), or—if it is a Rate-Responsive pacemaker— at either delivery 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 delivery time, but changed at runtime through the mode transition function.

We thus found it necessary to model four possible binding times for variations:

1) *Design-time binding.* In our method, this kind of variation is bound at modeling time and realized through statechart inheritance. For example, the BasePacemaker contains all the common functions for the product line, so the statecharts for these core behaviors were inherited by the other models in the product line. In addition, the other products, such as the RateResponsivePacemaker, add their own behaviors by adding new substates and new transitions to the inherited statechart .

2) *Configuration-time binding.* In our method, this kind of variation is bound at configuration time, which is the time right before animation when we choose among the classes we modeled which to use to generate the code.

For example, a pacemaker may have extra sensors to monitor a patient's motion, breathing, etc. We can either select or not select the ExtraSensor class when configuring the PL_Pacemaker. If ExtraSensor is selected, then the system is a RateResponsivePacemaker; otherwise, it is a BasePacemaker even though its statechart still models the behavior that would occur if ExtraSensor had been selected.

3) *Delivery-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, to model that a doctor can set the initial value of the cycle length parameter when a RateResponsivePacemaker is delivered to a patient, we use this technique.

4) *Run-time binding.* In our method, run-time variation is realized by the injection of different events during animation. This can be done manually or through a simulator that injects input signals at designated times or rates to simulate components external to the scope of the model triggering run-time parameter changes. Svahnberg and Bosch have described this additional aspect of binding, defining "internal binding" as occurring when the software contains the functionality to bind to a particular variant. "External binding," on the other hand, occurs when there is a person (such as a doctor) or tool that binds the variation [25]. As an example, 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.

Another way to bind variations at run time is through the use of the UML condition connector in the model. This constructs adds guards to the transitions. The guards are Boolean conditions that can be tested frequently during model execution. Depending on the current value of the condition, the statechart takes different transitions and thus goes to different states. This technique 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.

4.3.2. Representing dependencies. The dependencies among variations that we found needed 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 ModeTransitive 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 we found was 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.

4.4. Scenario-Guided Model Analysis

In this step we fully exercise the state model against the scenarios. We test the scenario against the state models, with one legal configuration of commonalities and variations enabled on such state models at a time ("legal" here means no violation of known dependencies). In order to promote reuse, we specify a generic sequence diagram for the product line first, and then customize it according to different configurations of variations.

A generic scenario includes all the components whose state models have been created in the second step are shown in the sequence diagram (see Fig. 4), each of which is represented as a separate instance line. This includes the components that shared by all the products (e.g., BaseSensor, PulseGenerator), and components that have variation models (e.g., PL_Pacemaker), components that are tied to certain products only thus are variations themselves (e.g., ExtraSensor), as well as black-box components that generate the required environment input in a real-time fashion (e.g., HeartSimulator and MotionSimulator).

The generic sequence diagram should also include the external events and internal events, representing messages generated from the system border of the sequence diagram and events that occur between the internal instances of the sequence diagram separately. The ordering of the internal messages should conform to the scenario specified in Step 1 of Section 3.1. Variations associated with the events, if possible, are also specified here. For example, in Figure 4, we use "n" in the event "evSimulateOn (SetHeartRate=n)" as a parameter that shows different heart rate. Though it is not a variation point of the cut-set component but of the black-box component, variations associated with data value can be specified in a similar way.

After the generic test scenario is specified, we customize it into different cases by adding variations until all the combinations of commonalties and variations indicated in the cut-set are covered. This customization should be done hand-in-hand with the state model customization such that if a certain variation is represented in the sequence diagram, it is enabled in the state model as well.

Thus, we execute the customized state model against its corresponding scenario. In this case study, we executed 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 of the state models, if their execution traversed the events in the right order as specified in their customized scenario, then the configuration passed the test.

4.5. Apply the Results to Enhance Safety

In the section we described three important uses of the results of this state-based modeling safety analysis.

1) ***Finding design errors.*** Any inconsistencies between the state model execution and the required scenarios, if not caused by incorrect modeling, reflected an error in the design that could eventually lead to a hazard. Mitigation involved first finding the causes of inconsistencies, then updating the design to remove the identified problem. For example, this configuration of variations may need to be specified as an illegal dependency, to inhibit the observed behavior. The model was then exercised against the same scenario again, until no inconsistency was found. Similarly, if any of the model executions conformed to a forbidden scenario that also showed a hazardous configuration of the variations where similar mitigation steps were needed.

2) ***Addressing safety-related concerns.*** The most important use of the results of the safety analysis here was to identify safety-related concerns in the software product line. For example, in this case study, we discovered from the PL-SFTA that the extra sensor (V3) was a potential single point of failure for the safety property S1. From the execution of the state model we found that was a credible threat that can actually happen. Through model execution with different combinations of variations enabled, we investigated a possible solution, which was to add a new safety property. 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.

3) ***Scoping models for formal verification.*** The result of our analysis provided useful information for further verification, as well. Exercising the state models makes evident which parts relate to a certain safety property. The scope needed for formal verification thus can be reduced. Also, since some of the potential design flaws are discovered and addressed in this step, the workload of verification is reduced. Finally, by specifying safety properties in terms of scenarios, it is relatively easy to update the scenarios in response to requirements changes and translation of the scenarios into temporal logics for formal verification. We plan to investigate our technique as a preliminary step for formal verification in future work.

5. Discussion

In the section we briefly discussed benefits and limitations with regard to tool support, scalability, and extensibility to other non-functional properties.

1) ***Limitations of the tool.*** Due to the limitation of the scenario description language used by the testing tool, or to limitations of the testing tool itself, some scenarios were not "testable". For example, the explicit timeout message cannot be tested by the TestConductor tool. Time intervals have to be between two messages sent from the system border to be testable. Another example is that Rhasody supports Live Sequence Charts [15] but TestConductor only supports UML sequence diagrams. However, the notion of a forbidden scenario is not explicitly supported by UML sequence diagrams. Thus, TestConductor does not have the built-in mechanism that allows us to specify certain pre-conditions that always lead to forbidden scenarios. This means that forbidden scenarios derived from hazard descriptions are actually tested in the same fashion as those required scenarios derived from safety properties.

2) ***Scalability to large product lines.*** Our method has only been applied to a simplified product line to date. We believe that it can be extended to analyze larger product lines because it uses the 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 the test scenarios and state models are done in a reusable fashion to reduce the workload as the number of system grows.

3) ***Extensibility to non-safety properties.*** In order to use our method for other properties (e.g., security), such properties must be able to be specified in terms of scenarios, and their related variations must be able to be identified in a scalable fashion, preferably hierarchically, as done in PL-SFTA. Any properties satisfying these requirements should be able to use our technique. Recent work to map concern space to software architectures will assist us in extending this approach to non-safety properties [18].

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 features and variations are added to the product line.

Acknowledgements

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

[1] Atkinson C, et. al., *Component-Based Product Line Enging with UML*, Addison-Wesley, 2002.

[2] Campbell L, et. al., "Automatically Detecting and Visualizing Errors in UML Diagrams", *Requirements Eng. Journal*, Springer-Verlag, Dec. 2002, pp. 264-287.

[3] Clauss M., "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.

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

[5] Czerny B. J., and M. Heimdahl, "Automated Integrative Analysis of State-based Requirements", In *Proc. 13th IEEE Int'l Conf. Automated Software Eng.* (ASE'98), Honolulu, HI, pp. 125-134.

[6] Dehlinger, J. and R. Lutz, "A Product-Line Approach to Safe Reuse in Multi-Agent Systems", In *ICSE 2005 4th Int'l Workshop on Software Eng. for Large-Scale Multi-Agent Systems* (SELMAS'05), St. Louis, MO, pp. 83-89, 2005.

[7] Dehlinger, J. and R. Lutz, "PLFaultCAT: A Product-Line Software Fault Tree Analysis Tool", *The Automated Software Eng. Journal*, to appear.

[8] Dehlinger J. and R. R. Lutz, "Software Fault Tree Analysis for Product Lines", In *Proc. 8th Int'l Symp., High Assurance Systems Eng.* (HASE '04), Tampa, FL, Mar. 2004, pp.12-21.

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

[10] Douglass B. P., *Doing Hard Time Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*, Addison-Wesley, 1999.

[11] Ellenbogen K.A. and M. A. Wood, *Cardiac Pacing and ICDs*, Blackwell Science, Inc., 2002.

[12] Feng, Q. and R. R. Lutz, "Bi-Directional Safety Analysis of Product Lines", In *Journal of Systems and Software*, Vol. 78, pp. 111-12, 2005.

[13] Gomaa, H., *Designing Software Product Lines with UML: From Uses Cases to Pattern-Based Software Architectures*, Addison-Wesley, 2005.

[14] Goseva-Popstojanova K, et. al., "Architectural-Level Risk Analysis Using UML", In *IEEE Trans. Software Eng.*, Vol. 29, No. 6, 2003.

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

[16] Kang K. C., et. al., "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", In *Annals of Software Eng.*, Vol. 5, pp. 143-168, 1998.

[17] Leveson N. G., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

[18] Liu J., Lutz R. and J. Thompson. "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. 4-8, 2005.

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

[20] Lutz, R. "Extending the Product Family Approach to Support Safe Reuse", In *The Journal of Systems and Software*, Vol. 53, No. 3, Sep. 2000.

[21] Lutz R. R. and R. M. Woodhouse, "Bi-directional Analysis for Certification of Safety-Critical Software", In *Proc. 1st International Software Assurance Certification Conference* (ISACC'99), Washington D.C., Mar. 1999.

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

[23] "Model Driven Development for Real-Time Embedded Applications", *Rhapsody Family Brochure*, http://www.ilogix.com/rhapsody/rhapsody.cfm, (Current May 2005).

[24] Schwanke R. and R. Lutz, "Experience with the Architectural Design of a Modest Product Family", *Software Practice and Experience,*Vol.34, pp. 1273-1276, 2004.

[25] Svahnberg M,, van Gurp J. and J. Bosch, "A Taxonomy of Variability Realization Techniques", *Groningen University Library*, http://www.ub.rug.nl/eldoc/dis/science/j.van.gurp/ (Current May 2005).

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