State-Based Modeling to Support the Evolution and Maintenance of Safety-Critical Software Product Lines

Jing Liu[1], Josh Dehlinger[1], Hongyu Sun[1] and Robyn Lutz[1, 2]
[1]*Department of Computer Science, Iowa State University*
[2]*Jet Propulsion Laboratory/Caltech*
{janetlj, dehlinge, sun, rlutz}@cs.iastate.edu

Abstract

Changes to safety-critical product lines can jeopardize the safety properties that they must ensure. Thus, evolving software product lines must consider the impact that changes to requirements may have on the existing systems and their safety. The contribution of this work is a systematic, tool-supported technique to support safe evolution of product-line requirements using a model-based approach. We show how the potential feature interactions that need to be modeled are scoped and identified with the aid of product-line software fault tree analysis. Further, we show how reuse of the state-based models is effectively exploited in the evolution phase of product-line engineering. To illustrate this approach, we apply our technique to the evolution of a safety-critical cardiac pacemaker product line.

1. Introduction

Changes to software requirements after deployment, due to system evolution, increase the difficulty of understanding, tracing, modeling and verifying the effects on system safety properties and can jeopardize the safety of the system [8]. Changes to the software requirements of a product line can greatly increase this difficulty because multiple systems are involved that may have varying safety properties and that can jeopardize the safety of the systems in different ways [6]. Safety-critical product lines, including cardiac pacemakers [20] [21], constellations of satellites [9] and medical imaging systems [28], need techniques and tools to accommodate and analyze the impact of system evolution on the product line and the product line's safety properties [17].

A product-line is a set of systems developed by a single company that share a common set of core requirements (i.e., the product line's commonalities) but differ amongst each other according to a set of managed variable requirements (i.e., the product line's variabilities) [27]. The utilization of product-line engineering for software systems is advantageous in that it exploits the reuse potential in the analysis, design and development of the commonalities and variabilities in each product-line member [30]. Studies suggest that product-line engineering can reduce development time and cost as well as increase the quality of products by a factor of 10 times or more [6]. Product-line evolution typically involves the addition of new features (i.e., variabilities) or the refining of existing variabilities (i.e., altering the allowed parameters of a product-line variability) [29].

Yet, product-line engineering still lacks the technical mechanisms to efficiently ensure the safety of each product-line system while fully taking advantage of its reuse potential [23]. Specifically, Kang [17] identifies the following as open problems for the viable use of product-line engineering:

- Verifying quality attributes (e.g., safety and reliability) and detecting feature interactions that may violate the quality attributes
- Modeling, analyzing and managing product-line features and feature interactions while avoiding the feature explosion problem
- Accommodating the evolution of the product line and adapting the product-line assets to the evolved requirements

The work described here addresses these problems in the context of the evolution and maintenance of a product line. Specifically, this work provides a structured, tool-supported decision mechanism, driven by the use of a product-line software fault tree analysis, to determine if new requirements, as a result of product-line evolution, can be safely integrated into the product line without introducing unchecked safety concerns. We utilize a product-line requirements analysis tool [25] and product-line software fault tree analysis tool [10] to augment and focus our state-based modeling of a product line on those new requirements

and potential feature interactions that may be safety-critical.

The contributions of this work are a tool-supported, state-based, safety analysis approach for the evolution of a software product line, including:

- Linking safety-critical, product-line requirements to their state-based model components
- Identifying and analyzing potential safety-critical feature interactions
- Modifying and reusing existing product-line state-based models to include new requirements from product-line evolution

This work is a part of a larger effort that investigates how safety-critical product lines evolve and that develops analysis techniques, tools and strategies to reduce the cost of safety analysis and enhance the safety and reusability of evolving product lines. The long-term goal is to provide safety analysis results for the new systems of a product line during requirements evolution in a timely and cost-efficient manner.

The remainder of this paper is as follows. Section 2 reviews related research in product-line engineering, state-based modeling for product lines and product-line safety analysis. Section 3 gives an overview of our approach to accommodate the safety analysis of evolving software product lines using state-based modeling. Section 4 details our technique using the evolution of a pacemaker as our safety-critical case study. Section 5 provides a brief discussion of our technique and our experience in its application. Finally, Section 6 provides some concluding remarks.

2. Background & Related Work

This work builds upon previous work integrating product-line engineering, state-based modeling and software safety analysis. Compared to our previous work in this field [20] [21], this work demonstrates how the safety analysis of a product line using a state-based modeling approach can accommodate product-line evolution.

2.1. Software Product-Line Engineering

The ability to reuse software engineering assets during system development continues to be of vital interest to industry as it offers the possibility to significantly decrease both the time and cost of software requirements specification, development, maintenance and evolution [27]. In product-line engineering, the common, managed set of features shared by all members, the commonalities, are reused for all members of the product line. For example, a commonality for a pacemaker is "*A pacemaker's pacing cycle length shall be the sum of the senseTime and the refractoryTime*".

The variabilities of a product line differentiate the product-line members and may have a design, configuration, delivery or run-time binding with the product-line member [20] [21]. For example, a run-time binding pacemaker variability is "*The senseTime of a pacemaker's pacing cycle may vary at run-time by setting the senseTime from 800 msec to 300 msec*".

Product-line dependencies restrict which combinations of variability subsets can form viable product line members. Dependencies may enforce safety requirements by preventing or restricting some feature interactions. For example, a pacemaker dependency is "*A modeTransitive type pacemaker must only use a 800 msec senseTime when it is operating in a Inhibited pacing mode*".

Product-line engineering is typically partitioned into two phases: domain engineering and application engineering [30]. A product line is initially defined by its commonalities and variabilities in the domain engineering phase. The benefits of product-line engineering come in the application engineering phase when the reusable assets defined in the domain engineering phase are exploited to create product-line members. Product-line evolution typically involves the addition of new features (i.e., variabilities) or the refining of existing variabilities (i.e., altering the allowed parameters of a product-line variability) [29]. For instance, a requirement evolution for the pacemaker variability given above may expand allowable senseTime pacing cycle to also include some value between 800 msec and 300 msec, e.g., 500 msec.

2.2. Model-Based Software Development

State-based modeling has previously been used as a mechanism to detect the correctness of the requirements and design as well as to aid in the verification of behavioral requirements [1], [7]. Harel and Marelly, like us, have used a scenario-guided approach to testing state-based models as a way to identify missing requirements [16]. However, their work concentrates on validating the safe behavior of single systems, whereas the work described here aims at validating the safe behavior of the multiple systems within a product line.

Software product lines have been modeled in various ways using extensions of UML to aid in the reuse of UML assets. For example, Clauss extends UML to support features diagrams as well as extending the package diagram to incorporate variabilities descriptions [5]; Doerr classifies the relationships within a variation model and relates them to UML

notation [12]; Gomaa uses executable UML statecharts to as a product-line model [15]; and Prehofer uses state-model composition to evaluate the interaction of features [26]. The work described in this paper also uses executable UML but focuses on providing assurance to the satisfaction of the safety properties of the product line as well as examining the potentially unsafe feature interactions.

More recently, Deng, Lenz and Schmidt have demonstrated a model-transformation approach using the Domain Specific Modeling Language to address the changes in a product line's architecture as a result of domain evolution [11]. Our work concentrates on the impact of software evolution on the safety of the system, rather than on the architectural impact.

2.3. Software Safety Analysis

Safety analysis for software product lines is still immature. Safety analysis approaches have been proposed to verify safety properties and discover missing safety requirements for the multiple systems of a product line. Feng and Lutz [14] propose a bi-directional approach that uses a forward search to discover the effects of a hazard coupled with a backward search from faults to their contributing causes to verify and discover safety requirements. Lu and Lutz propose a failure contribution analysis for product lines to help the analysis of the contributions of commonality and variability trees to root node hazards [22]. Yet, these two approaches rely on a static analysis of the product-line requirements rather than the executable analysis done in this work.

This work utilizes two tool-supported product-line safety analysis methods to support the creation of state-based models and to analyze the evolution and feature interactions of product-line requirements.

DECIMAL is a product-line requirements analysis tool that documents the commonalities, variabilities and dependencies of a product line during the domain engineering phase [25]. During the application engineering phase, DECIMAL verifies that the selection of variabilities for a product-line member do not violate the product line's prescribed dependencies.

PLFaultCAT is a tool that aids the construction and analysis of product-line software fault tree analyses (SFTA) [10]. 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 [18]. PLFaultCAT allows engineers to construct the product-line SFTA and associate the commonalities and variabilities, from DECIMAL, with the leaf nodes of the SFTA in the domain engineering phase. During application engineering, PLFaultCAT semi-automatically produces the product-line members' SFTAs from the product-line SFTA.

The work reported here, as in our previous work [20] [21], uses executable UML within the Rhapsody software modeling environment as well as the TestConductor tool by I-Logix [24].

3. Approach

This section describes the construction of the safety analysis of an evolving software product line using state-based modeling. It focuses on how to identify, model and analyze potentially unsafe feature interactions.

3.1. Safety Analysis of Evolving Software Product Lines Using State-Based Modeling

We here provide a step-by-step overview of our technique for safety analysis of software product lines using state-based modeling for a product line during evolution.

3.1.1. Commonality and Variability Analysis. The Commonality and Variability Analysis (CVA) documents the product line's requirements [30]. During evolution, new feature requirements (i.e., variabilities) are added to the CVA, possibly using a product-line requirements analysis tool, such as DECIMAL [25], as done here.

3.1.2. Product-Line Software Fault Tree Analysis (SFTA). A product-line SFTA will need to accommodate the new features if they can potentially contribute to causing one of the failures described in the SFTAs. The new features may require the modification of the product-line SFTA by adding entirely new fault trees as a result of the possibility of new root node hazards occurring. This requires the construction of a product-line SFTA just as done during the initial development of a product line [10].

Additionally, new features introduced during product-line evolution may need to be included in existing product-line SFTAs. To accomplish this, each existing fault tree is analyzed to see how the new feature(s) can contribute to cause the root node hazards. This may entail adding subtrees to the existing fault trees or associating the requirements of the new feature with the leaf nodes of the fault tree. Here we use the SFTA tool PLFaultCAT [10], to achieve this.

3.1.3. Variation model generation. We map the leaf nodes of the product-line SFTA to architectural components and then model the behavior of the

architectural component in a state model. During the initial development of a product line, the state-chart model is incrementally built from the product that has the fewest variable features until all features are included into the state model [20] [21].

To address product-line evolution, any new features are incrementally integrated into the state model. To achieve this, any newly created SFTAs, a result of Step 2, will need to map the SFTA's leaf nodes to a new or existing architectural component. If they are mapped to an existing component, that component's behavior must be modified to include the new behavior introduced by the new feature(s). If they are mapped to a new architectural component, that new component's behavior should then be modeled and integrated into the product-line state model. For the existing product-line SFTAs that were modified to accommodate the new features, we need to include the new behaviors into the architectural components representation in the state model.

3.1.4. Scenario derivation. Using the product-line SFTA, we derive required scenarios (i.e., those scenarios that enforce a safety property) and forbidden scenarios (i.e., those scenarios that emulate a hazard). For the newly created product-line SFTAs, the process described in [20] [21] suffices. For the existing product-line SFTAs that were modified as a result of the new features, the scenarios that were developed during the initial product line's construction must be altered to accommodate the behavior described in the new subtrees of the SFTAs. This will result in modified testable scenarios that need to be re-executed in Step 5 since the behavior they display as a result of evolution may differ from when they were executed and verified during the product line's initial development.

3.1.5. Scenario-guided model analysis. The developed scenarios, from Step 4, are exercised against the state model, from Step 3. Although the introduction of the new feature as a result of evolution may not have altered all the testable scenarios from the initial development, all scenarios should be exercised against the model to ensure that the inclusion of new features' behavior into the model does not produce undesired/unknown effects (i.e., a regression testing approach). This step then follows [20] [21]: a failure in the execution of the required scenarios indicates inconsistencies between the model execution and the specified scenario; a forbidden scenarios execution will indicate a need to update the design if it is found to allow illegal/hazardous behavior. In each case, an update to the design is warranted if undesired behavior is detected when executing the scenarios in the state model. In this work, we used TestConductor to exercise the model in the Rhapsody modeling environment [24].

3.2. Identifying and Modeling Safety-Critical Feature Interactions

The evolution of a software product line is more complex than for single software systems since new, possibly conflicting, features from the existing products in the product line and the newly introduced features for the new products can result in unsafe/undesirable feature interactions [29]. For example, the 1996 explosion of the initial flight of the Ariane 5 rocket was partially blamed on the interaction of the new features introduced in the Ariane 5 with the features retained from the earlier, Ariane 4, rocket [19]. Thus, it is crucial to ensure that product-line evolution does not introduce feature interactions that compromise the safety properties that the product line previously ensured.

To address this, our approach focuses on the identification and modeling of safety-critical feature interactions to determine whether they may cause a hazard. In the case that the feature interactions could cause a hazard, we explore in simulation the effects of possible alternatives in the model to prevent such an unsafe feature interaction. The identification and analysis of new safety-critical feature interactions in the product line introduced as a result of the inclusion of a new feature(s) during evolution consists of the steps described below.

3.2.1. Identification of safety-critical feature interactions. A product-line software fault tree analysis (SFTA) associates a product line's requirements (i.e., commonalities and variabilities) with the leaf node failure events that may lead to the occurrence of the root node hazard. As described in Section 3.1, Step 2 as well as in [10], the evolution of a product line will associate new product-line requirements with the leaf nodes of existing SFTAs along with former requirements.

After the adaptation of the product-line SFTAs to the new features introduced as a result of evolution, the safety-critical feature interactions can be identified by searching for those product-line requirements that frequently contribute to the possible causes of the fault tree's failure nodes. PLFaultCAT [10], the product-line SFTA tool used here, can automatically identify those product-line requirements and combination of product-line variabilities (i.e., features) that contribute to the most potential failures as defined in the SFTA.

This analysis provides a prioritized list of those product-line requirements and feature interactions that warrant further scrutiny using an executable state-based model. That is, those product-line requirements and feature interactions that are deemed to contribute to the most fault tree failure nodes are more likely to have unsafe interactions with existing product-line requirements and should have their behaviors modeled in order to determine the safe/unsafe behaviors using a dynamic analysis.

3.2.2. State-based modeling of feature interactions to determine safe/unsafe behavior. We here describe our approach using executable state models and scenarios. To determine the safety of feature interactions using our state-based model, we first take a manageable sub-tree of the product-line SFTA. The variabilities in the cut-set of such a fault tree can be used to map to components in the architecture diagram of the product line. If there are new features introduced into the product line, we need to update the architecture design if such a new feature will introduce new components or new associations between components.

Next, we take the state models of those components where the safety related variabilities reside. For newly introduced features, it is likely that the existing state models will be updated, or new state models will be created. We then derive the events that are identified as potentially causing hazards and their direct consequences from SFTA. The causative events and their consequences, represented as message passing between involved components, form required or forbidden scenarios to analyze in the following steps.

Next, we execute the models and inspecting the execution sequences either manually or automatically, with the scenarios previously identified as guidance. The manual inspection includes monitoring the message-passing sequence diagram among the components that are identified in the derived scenarios, pausing at points of importance, and selectively investigating details in the animated statechart view of a specific component when necessary. Manual inspection also includes injecting events at run-time to test the response of the system under different environmental inputs.

The automatic inspection involves using a scenario-based state model testing tool, such as TestConductor, that captures requirements regarding absolute or partial ordering of messages as sequence diagrams, and testing the sequence diagram against the actual order of message-passing during the state model execution. Such tests can be automatically executed for improved inspection.

The manual inspection is more flexible and more informative for requirements that cannot be easily modeled by sequence diagrams due to the nature of the requirements or limits of the tool [20] [21], while the automatic inspection is more thorough, providing more assurance regarding a testable scenario. The outcome gives users information regarding whether a forbidden scenario is likely to happen and how it may happen, or whether a required scenario can sometimes not happen. This is because the execution will give the actual execution scenarios providing details confirming or contradicting the scenarios derived. Note that this scenario-guided inspection of model execution gives no guarantee as to whether a required scenario is always going to happen or a forbidden scenario is never going to happen – that requires the more rigorous reasoning provided by formal methods [4].

After inspection, we need to find mechanisms to avoid forbidden scenarios from happening or enforcing required scenarios, using the detailed results from the previous step as guidance. Such mechanisms, once implemented in the state model, will again be inspected during execution, as described above, to decide if they do achieve their goals. Once confirmed, these mechanisms can be used to suggest new requirements update.

While a new feature is likely to update an existing SFTA or even introduce a new SFTA, previous SFTA-related state models may be re-validated (by running through the process described above) to ensure that the new feature does not interfere with them. Such a re-validation process can be done by adding the new feature related component into the scenario to check if there is any potential interaction between this component and the components residing in the original sequence diagram.

4. Application to the Evolution of a Product-Line Pacemaker

This section applies the approach described in Section 3 to a safety-critical, product-line cardiac pacemaker.

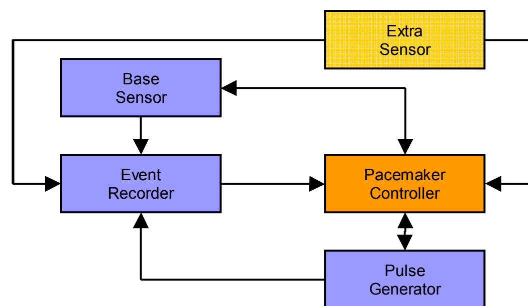

Figure 1. Product-Line Architecture after Evolution

4.1. Description and Evolution

To illustrate our approach, we build upon the pacemaker product line described in [20] [21]. 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. It 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 [13]. In our simplified example, the monitoring device has three basic components: a sensing component (sensor) that senses heart beat, a stimulation component (pulse generator) that generates pulses to the heart, and a controlling component (controller) that configures different pacing and sensing algorithms and issues commands. Here, we only consider a single-chambered product line of pacemakers that does pacing and sensing in the heart's ventricles.

Our simplified pacemaker product line consists of the following products and features:

- **BasePacemaker** – This product has the basic functionality shared by all pacemakers: generating a pulse whenever no heart beat is sensed during the sensing interval.
- **ModeTransitivePacemaker** – This product can switch between InhibitedMode and TriggeredMode during runtime. In the InhibitedMode, the pacemaker acts exactly like a BasePacemaker. In the TriggeredMode, a pulse follows every heartbeat to provide a different type of therapy.
- **RateResponsivePacemaker** – This product acts similarly to the BasePacemaker but contains an extra sensor allowing it to adjust its sensing interval according to the patient's current activity level: LRLrate, for a patient's normal activities and URL rate, for when a patient is exercising.
- **ModeTransitive-RateResponsivePacemaker** - This product combines the features of the ModeTransitivePackemaker and the RateResponsivePacemaker.

The evolution of the pacemaker product line that we consider here involves the addition of an EventRecorder component, shown in Fig. 1, to log critical events in the major components of a pacemaker and is used for making therapy decisions. For instance, EventRecorder calculates the number of heart beats sensed by BaseSensor during a fixed recording interval and compares that value with some threshold value to decide if the pacemaker should switch between InhibitedMode and TriggeredMode during run-time. Different pacemakers can log different events at different times, as shown in Table 1.

The addition of the EventRecorder feature was included into the product line's requirements using DECIMAL [25].

Due to the cross-cutting nature of the EventRecording feature, the risk of unsafe feature interaction is higher. For example, when the average number of heart beats in a 6000 msec recording interval exceeds a 24-beat threshold, the EventRecorder shall consider that the patient's heart is fibrillating, so it will command the PacemakerController to switch from InhibitedMode to TriggeredMode to defibrillate it. It must be ensured that the features added to PacemakerController due to the introduction of EventRecorder interact with the existing features in PacemakerController in a predictable manner and that there are no unexpected and/or unsafe feature interactions.

Table 1. Event Recording Feature's Commonality and Variability

Product Name	Component Name	Events to Log
Base Pacemaker	Base Sensor	Average heart rate sensed every fixed recording interval
	Pulse Generator	The pulse width of every pulse being made
Mode Transitive Pacemaker	Base Sensor	Average heart rate sensed every fixed recording interval
	Pulse Generator	1) In the Triggered mode, the average number of pulses generated every fixed recording interval 2) In the Inhibited mode, the pulse width of every pulse being generated
Rate Responsive Pacemaker	Base Sensor	Average heart rate sensed every fixed recording interval
	Pulse Generator	The pulse width of every pulse being made
	Extra Sensor	The percentage of the pacemaker sensing at LRLrate every fixed recording interval

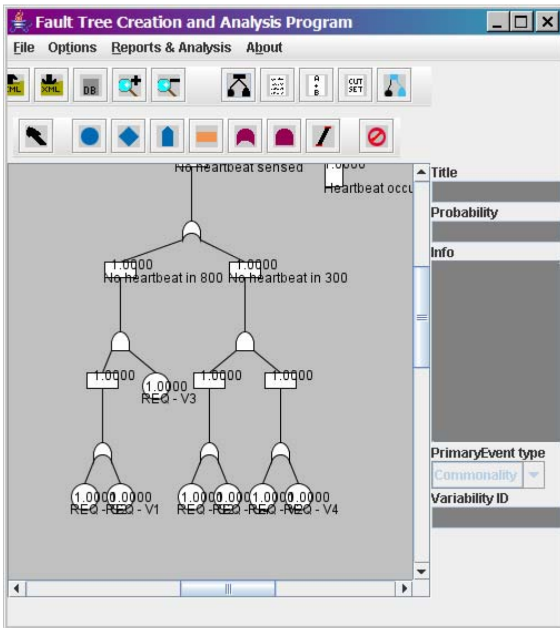

Figure 2. Excerpt of Pacemaker Product-Line SFTA in PLFaultCAT after Evolution

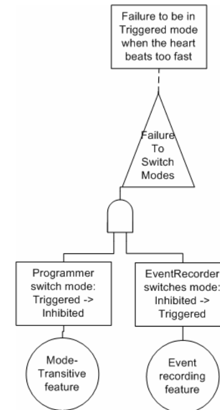

Figure 3. Excerpt of Product-Line SFTA Illustrating Potential Feature Interactions

4.2. Product-Line SFTA Evolution

The inclusion of the EventRecorder feature into the pacemaker product line required both the updating of existing product-line software fault tree analyses (SFTA) and the creation of new product-line SFTAs to accommodate the new failure modes that the new feature brings to the product line. For example, because of the new behavior introduced by the EventRecorder feature, a product-line SFTA with a root node of “Failure to switch modes”, shown in Fig. 2, had to be added. The creation of the new product-line SFTA with a root node of “Failure to switch modes” required the association of the requirements of the new EventRecorder feature as well as those features from existing product-line products. For example, as illustrated in Fig. 3, the ModeTransitive feature (found in the ModeTransitivePacemaker and the ModeTransitive-RateResponsivePacemaker products) may interact with the EventRecorder feature to cause a hazard. Yet, from examining the SFTA, it is not entirely clear how these two features can interact to cause such hazards, thus the need for further analysis.

Using PLFaultCAT, we can analyze the set of product-line SFTAs to find other such combinations of features that may cause hazards to direct the safety analysis, described in Section 4.3, to those feature interactions, like shown in Fig. 3, which may need to be further scrutinized.

4.3. State-Based Modeling of Safety-Critical Feature Interactions and Derivation of New Safety Requirements

This section illustrates the steps involved in using the state-based modeling approach to promote safe evolution of a product line. It uses the example of an EventRecording feature introduced as an existing pacemaker product line evolved.

The new EventRecording feature has introduced a possible hazard, “failure to be in the TriggeredMode when the heart beats too fast”, shown in Fig. 3. The subtree shown here concretizes this high-level hazard by adding events (e.g., the mode switch event sent from the operator and the mode switch event sent from the EventRecorder), conditions (e.g., the heart beats too fast), and the consequences (both safe and unsafe, e.g., required scenario: remains in TriggeredMode; forbidden scenario: fails to remain in TriggeredMode). The refinement of the hazard node in this way forms the scenario to check against the state models.

For the subtree in Fig. 3, we model the components that implement the leaf node requirements. For example, the ModeTransitive feature is implemented by the PacemakerController component, the MotionSimulator component, and the ExtraSensor component.

After the state models are generated, we instantiate the scenario captured in the subtree by mapping the events, conditions, and consequences to model-level elements. For example, the mode switch event sent from the operator is mapped to the “evInhibitedMode()” message, and the mode switch event sent from the EventRecorder is mapped to the “evTriggeredMode()” message, while the “heart beats too fast” condition is represented by a concrete threshold for the heart beats (16 beats while in

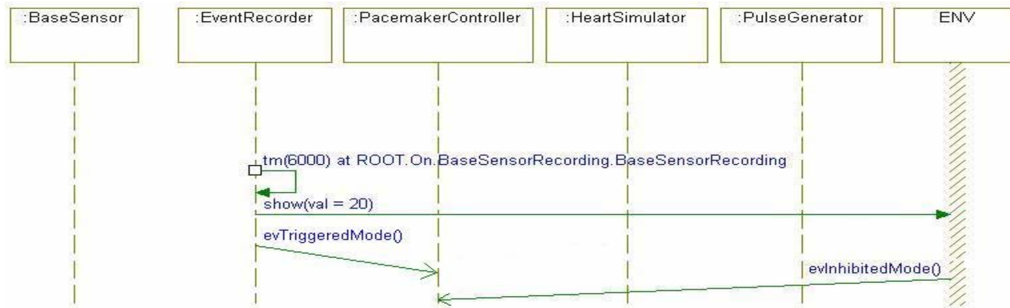

Figure 4. Excerpt of Pacemaker Animated Sequence Diagram

LRLrate, and 24 beats while in URLrate). The safe and unsafe consequences are mapped to PacemakerController component being in the TriggeredMode state and the InhibitedMode state, respectively.

Note that simple state models of other components, even if they do not directly implement the leaf node requirements, such as the PulseGenerator component, can also be generated if their responses in the execution help illustrate the above scenarios more clearly.

The next step animates the generated state models using Rhapsody. The animation process, as explained in Section 3.2, is mainly composed of animated sequence diagrams illustrating message passing during model execution, and animated state charts illustrating states and transitions taken at run time.

Fig. 4 shows a portion of the animated sequence diagram. It is a point where the fixed recording interval has expired (as shown by the "tm(6000)" message), Since the recorded number of heart beats is 20 (greater than the 16 threshold, indicating that the heart is beating too fast), the EventRecorder commands PacemakerController to switch mode from InhibitedMode to TriggeredMode. However, if we then inject an evInhibitedMode() event to PacemakerController, it will switch back to Inhibited mode, as shown in the animated statechart in Fig. 5 (the current states are highlighted), *despite the fact that the heart is still beating too fast.*

The animation shows that the scenario we captured in the fault tree, Fig. 3, and instantiated in the model level can actually happen. It also shows how this unsafe scenario can happen: when the two events (evInhibitedMode() and evTriggeredMode()) occur in a certain order (evTriggeredMode() first and evInhibitedMode()) second). As we discuss below, it is these sorts of ordering and timing issues that the executable state model, unlike the fault tree, can reveal.

Since the animation is done with the executable models, it also provides concrete insights into how to mitigate this potentially hazardous scenario, namely by adding a mechanism that locks the PacemakerController in Triggered mode when the heart beats too fast. The benefit of model-level analysis is that the new mechanism can be tested right away to see if it conforms to the safe scenario.

Another benefit is that we can readily investigate several mechanisms in order to select the more reliable and easy-to-implement one. For example, there are at least two ways that we can implement the mitigation mechanism. One option is to name the mode-switch messages sent from the EventRecorder and the Programmer differently and to give the EventRecorder's message higher priority than the Programmer's. Another option is to set up an internal variable in the PacemakerController that records the heart's status as beating too fast or not. Such a variable is used for guarded transitions from the TriggeredMode state to the InhibitedMode state and can only be changed when the EventRecorder detects a heartbeat.

While both mechanisms prevent the unsafe scenario from happening, the first one is more restrictive in that it grants the EventRecorder priority on messages switching both into, and out of, TriggeredMode The second one just enforces the EventRecorder's priority in switching out of TriggeredMode. However, the second alternative allows the possibility of the Programmer and the EventRecorder racing to switch into TriggeredMode. If the second alternative is selected, this suggests that we may want to introduce additional requirements to handle this possible race condition.

5. Discussion

This work utilized a product-line software fault tree analysis (SFTA) and state-based modeling of critical components to identify potentially unsafe feature interactions [5]. This approach provides some advantages over the use of feature diagrams. Feature diagrams can document identified interactions but fail to indicate the feature interactions that are safety-critical. The product-line SFTA, however, aids us in identifying those feature interactions that can cause

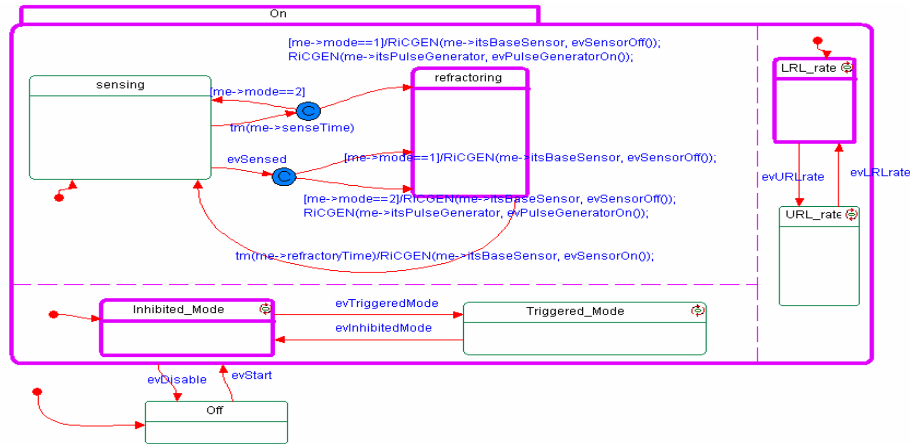

Figure 5. Animated PacemakerController Statechart in the Evolved Product-Line State Model

top-level failure events (i.e., those feature interactions that are safety-critical). We found that using the product-line SFTA greatly reduced the number of feature interactions that had to be investigated. Focusing on critical interactions makes our approach more amenable to use in an industrial setting.

The use of a state-based modeling approach for safety analysis during evolution is advantageous because it can both build on and extend the product-line fault tree analysis. Unlike SFTA, an executable state-based model can analyze and model the timing/ordering of failure events to determine their possible safety implications. In addition, we found that because the SFTA is a static asset, it lacks the ability to animate and explicitly show how a safety property may be violated. The use of an executable state-based model, however, allows the simulation of the behaviors described by the requirements in the fault tree to illustrate the violation of a safety property.

We found that although Rhapsody's executable state-based models support real-time notions, as required in our pacemaker case study, it cannot enforce exact real-time measurement. Thus, the state-based modeling technique described here is not suitable for testing border time values; rather, it can handle testing the ordering logic and relative timing of failure events.

The exploration of alternative new software behaviors in the state-based model to prevent or mitigate the violation of a safety property allows immediate feedback on whether proposed, new safety requirements will indeed guard against the violation of the safety property. Moreover, the executable state-based model, unlike a SFTA, can explore multiple solutions to come up with a reliable and easy-to-implement mitigation strategy. This then drives the updating of the product line's requirements to include the new safety requirements. Such feedback is impossible to ensure using the SFTA alone. Thus, the inclusion of a state-based modeling safety analysis approach, as described here, may improve the safety case that a safety-critical product line must make when requiring certification from an outside governing body.

Reuse of the product-line SFTA as well as of the product-line state-based models constructed during the initial development of the product line, occurred during system evolution in this work. Although, some updates were required to accommodate the new features introduced, large parts of the previously developed safety analysis assets could be reused.

The use of DECIMAL [25], the product-line requirements documentation and analysis tool, coupled with the use of PLFaultCAT [10], the product-line SFTA tool, improved the traceability of the requirements to the components of the state-based models. The leaf nodes of the fault trees constructed in PLFaultCAT are associated with the commonalities and variabilities of the product line, and the state-based models are derived using information from the SFTA. Additionally, the use of PLFaultCAT to identify those feature interactions that may be safety-critical, and therefore should be analyzed using state-based models, helps maintain the traceability of requirements to the state-based safety analysis as the product line evolves.

6. Conclusion

Product-line engineering presents an advantageous approach to developing software systems because the reuse can reduce the development time and cost. Yet, handling product-line evolution is more complex than in traditional software systems because changes to the software requirements may affect or even compromise the various safety properties of multiple products. In particular, the analysis of feature interactions is important because, during evolution, the new features introduced into a product line may have unknown and unsafe interactions with the existing features.

This paper illustrated an approach, built on our previous work with stable product lines, to performing a safety analysis on an evolving product line using a product-line software fault tree analysis to direct state-based modeling. The paper detailed and demonstrated a tool-supported technique to: 1. link product-line requirements to their state-based model components; 2. identify and analyze safety-critical feature interactions; and 3. modify and reuse product-line state-based models to analyze the new features added as a result of evolution. This technique utilized a product-line software fault tree analysis to avoid and manage the complexity of feature interactions.

Future work includes refining our technique and developing further tool-support to provide further guidance in the application of this technique.

7. Acknowledgements

This research was supported by the National Science Foundation under grants 0204139, 0205588 and 0541163. We also thank Telelogic for the use of I-Logix's Rhapsody and TestConductor tools.

8. References

[1] Bennett, K. and Rajlich, V., "Software Maintenance and Evolution: A Roadmap", *The Future of Software Eng.*, ACM Press, pp. 75-87, 2000.

[2] Booch, G., et. al.., *The Unified Modeling Language User Guide*, Addison-Wesley, 2005.

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

[4] Clarke, E. M. and Wing, J. M., "Formal Methods: State of the Art and Future Directions", *ACM Computing Surveys*, 28(4):626-643, 1996.

[5] Clauss M., "Modeling variability with UML", *Proc. Net.ObjectDays 2001 Workshop on Generative and Component-Based Software Eng.*, pp. 226–230, 2001.

[6] Clements, P. and Northrup, L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[7] Czerny B. J., and Heimdahl, M., "Automated Integrative Analysis of State-based Requirements", *Proc. 13th Int'l Conf. Automated Software Eng*, pp. 125-134, 1998.

[8] de Lemos, R., "Safety Analysis of an Evolving Software Architecture", *Proc. 5th Int'l Symp. on High Assurance Systems Eng.*, pp. 159-167, 2000.

[9] Dehlinger, J. and Lutz, R. R., "A Product-Line Approach to Promote Asset Reuse in Multi-Agent Systems," *Software for Multi-Agent Systems IV*, Lecture Notes in Computer Science 3914, pp. 161-178, 2006.

[10] Dehlinger, J. and Lutz, R. R., "PLFaultCAT: A Product-Line Software Fault Tree Analysis Tool," *Automated Software Eng. Journal*, 13(1):169-193, 2006.

[11] Deng, G., et. al., "Addressing Domain Evolution in Model-Driven Software Product-Line Architectures", *Proc. Workshop Model-Driven Development for Software Product Lines: Fact or Fiction?*, 2005.

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

[13] Ellenbogen, K. A. and Wood, M. A., *Cardiac Pacing and ICDs*, Blackwell Publishing, 2005.

[14] Feng, Q. and Lutz, R. R., "Bi-Directional Safety Analysis of Product Lines," *Journal of Systems and Software*, 78(2):111-127, 2005.

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

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

[17] Kang, K., "Software Product Line Research Topics," *Proc. 10th Int'l Software Product Line Conf.*, 2006.

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

[19] Lions, J. L., "Ariane 5: Flight 501 Failure Report", http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, (Current January 2007).

[20] Liu, J., Dehlinger, J. and Lutz, R. R., "Safety Analysis of Software Product Lines Using State-Based Modeling," To appear *Journal of Systems and Software.*

[21] Liu, J., Dehlinger, J. and Lutz, R. R., "Safety Analysis of Software Product Lines Using State-Based Modeling," *Proc. 16th IEEE Int'l Symp. on Software Reliability Eng.*, pp. 21-30, 2005.

[22] Lu, D and Lutz, R. R., "Fault Contribution Trees for Product Families", *Proc. 13th Int'l Symp. Software Reliability Eng.*, pp. 231-242, 2002.

[23] Lutz, R. R., "Software Engineering for Safety: A Roadmap," *Proc. Conf. on the Future of Software Eng.*, pp. 213-226, 2000.

[24] "Model Driven Development for Real-Time Embedded Applications", *Rhapsody Family Brochure*, http://www.ilogix.com/rhapsody/rhapsody.cfm, (Current January 2007).

[25] Padmanabhan, P. and Lutz, R. R., "Tool-Supported Verification of Product-Line Requirements", *Automated Software Eng. Journal*, 12(4):447-485, 2005.

[26] Prehofer, C., "Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams", *Proc. 7th Int'l Workshop Feature Interactions in Telecommunications and Software Systems*, 2003.

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

[28] Schwanke R. and Lutz, R., "Experience with the Architectural Design of a Modest Product Family", *Software Practice and Experience*, 34(13):1273-1296, 2004.

[29] Svahnberg, M. and Bosch, J., "Characterizing Evolution in Product Line Architectures", *Proc. of the 3rd annual IASTED Int'l Conf. on Software Eng. and Applications*, pp. 92-97, 1999.

[30] Weiss, D. M. and Lai, C. T. R., *Software Product-Line Eng.*, Addison-Wesley, 1999.