2008

Safety analysis of software product lines using state-based modeling and compositional model checking

Jing Liu
Iowa State University

Safety analysis of software product lines using state-based modeling and compositional model checking

by

Jing Liu

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Robyn R. Lutz, Major Professor
Samik Basu
Carl K. Chang
Suraj C. Kothari
Gary T. Leavens
Arun K. Somani

Iowa State University

Ames, Iowa

2008

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

The Ph.D. study is a great learning and self-discovery experience. I owe my gratitude to a great many people who helped me through this journey.

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Robyn R. Lutz, for all the wisdom, sight, care, and encouragement more than I could ever expect from an advisor. She led me into the field of safety-critical product lines, and passed on her passion on scientific research. I will always treasure the experience working with her.

I would also like to offer my heartfelt thanks to all the members of my doctoral committee: Dr. Samik Basu, Dr. Carl Chang, Dr. Suraj Kothari, Dr. Gary Leavens and Dr. Arun Somani. I am truly fortunate to have such an exceptional committee. Their support and expertise greatly helped me improve my dissertation work. Special thanks go to Dr. Samik Basu, with whom I closely collaborated for a significant part of this research, and Dr. Gary Leavens, for introducing me to the fascinating world of the semantics of programming languages.

I am grateful to all my co-authors: Dr. Samik Basu, Dr. Josh Dehlinger, Dr. Birgit Geppert, Miriam Hauptman, Dr. Robyn Lutz, Dr. Hridesh Rajan, Dr. Frank Rößler, Hongyu Sun, and Dr. Jeffrey Thompson. Most of the work present in this dissertation are the fruits of the collaborations with them. Their strong research skills and work ethics inspired me to become a better researcher. Special thanks go to Dr. Josh Dehlinger, for helping me grasp a key part of this dissertation work, and Dr. Jeffrey Thompson and Dr. Birgit Geppert, for mentoring me in industry research.

My special thanks also go to many other researchers I have worked with: Qian Feng, Dr. Arka Ghosh, Dr. Lars Grunske, Youssef Hanna, Dr. Aaron Larson, Juri Memmert, Dr. José Merseguer, Dr. Kyle Nelson, Dr. Matthias Riebisch, and Dr. David Weiss. Their thoughtful and insightful opinions provided many ideas for me to advance my research work.

This research would not have been possible without the generous support from numerous people and organizations. Ben Di Vito sent us the SMV code for SAFER. Telelogic (formerly I-Logix) granted us the free use of Rhapsody and TestConductor tools. Cadence Berkeley Labs provided us the Cadence SMV tool. National Science Foundation (grants 0204139, 0205588, 0541163), the Department of Computer Science, and Iowa State University presented me both the opportunity and financial means to explore the research topics I truly love. Moreover, our lab's collaboration with Boston Scientific and Avaya Labs Research enabled us to think of the research problems and solutions in an industrial setting.

I would like to especially thank all the past and present members of the Laboratory for Software Safety: Sarah Chapman, Josh Dehlinger, Qian Feng, Miriam Hauptman, Meredith Humphrey, Sandeep Krishnan, Dinanath (Yogesh) Nadkarni, Barbara Nsiah, Kendra Schmid, Jonathan Schroeder, Hongyu Sun, Lada Suvorov, Oko Swai, Erika Willi, and Wei Zhang. Together they build a supportive and active environment to succeed.

I have received great support from many people in the Department of Computer Science: Jerry Anderson, Dr. Ying Cai, Gloria Cain, Yuansen (Jason) Chen, Maria-Nera Davis, Linda Dutton, Dr. Shashi Gadia, Dr. Jack Lutz, Cindy Marquardt, Dr. Hridesh Rajan, Jim Schlosser, Dr. Wallapak Tavanapong, and Dr. Johnny Wong. Their knowledge and friendliness to assist the students really brightened my day. In particular, I would like to thank Linda Dutton. She has been there for every milestone in my graduate student life. Dealing with the administrative paperwork became pleasant because of her.

I am also thankful to Dr. Susan Jarnagin and Joe Struss, for providing all the helpful resources to make my thesis writing much more pleasant; to Sara Harris and Laurel Tweed, for helping arrange my defense; and to Virginia McCallum, Dilok Phanchantraurai and Steve Kravinsky, for making my life as an international student much easier.

Many friends have been there with me through the up and downs in graduate-student life. Among them, special thanks to Yu Cao, Josh Dehlinger, Qian Feng, Xiaoyang Gu, Xin Guo, Youssef Hanna, Ru He, Girish Lingapa, Danyu Liu, Dingding Lu, Mary Loomer, Hua Ming, Sule Karaman, Chuang Wang, Fengming Wang, Youdan Wang, Ge Xu, Lijun Yan, Meng Yang, and Peimin Zhang.

Most importantly, I am indebted to my family. To Hongxian Ma, Wenlin Liu, Kefei Wei, Yixu Ma, Yizhu Song, Shiqing Liu, Ronghua Diao, Chunzhe Luo and Qiqun Wei: I cannot imagine coming to this milestone without your unwavering love and support to me. This dissertation is dedicated to you.

ABSTRACT

Software product lines are widely used due to their advantageous reuse of shared features while still allowing optional and alternative features in the individual products. In high-integrity product lines such as pacemakers, flight control systems, and medical imaging systems, ensuring that common and variable safety requirements hold as each new product is built or existing products are evolved is key to the safe operations of those systems.

However, this goal is currently hampered by the complexity of identifying the interactions among common and variable features that may undermine system safety. This is largely due to (1) the fact that the available safety analysis techniques lack sufficient support for analyzing the combined effects of different features, and (2) existing techniques for identifying feature interactions do not adequately accommodate the presence of common features and results in repeated checking across different products.

The work described here addresses the first problem by systematically exploring the relationships between behavioral variations and potential hazardous states through scenario guided executions of the state model over the variations. It contributes to a solution to the second problem by generating formal obligations at the interfaces between features, so that sequentially composed features can be verified in a way that allows reuse for subsequent products.

The main contributions of this work are an approach to perform safety analysis on the variations in a product line using state-based modeling, a tool-supported technique that guides and manages the generation of model-checkable properties from product-line requirements, and a formal framework for model checking product-line features that removes restrictions on how the features can be sequentially composed. The techniques and their implementations are demonstrated in the context of a medical-device product line.

CHAPTER 1. INTRODUCTION

Software product lines are widely used due to their advantageous reuse of shared elements, but this reuse across different products poses challenges for safety analysis of product lines. Especially for high-integrity product lines, we would like to ensure that key safety requirements hold in every product as they are developed and evolved. However, existing safety analysis techniques for product lines are currently hampered by the difficulty of managing variations and their potential interactions across an entire product line, as well as by the complexity of integrating the analysis results of the variations in a way that allows effective reuse.

The work described in this dissertation contributes to a solution by constructing a reusable safety analysis framework for safety-critical software product lines. The framework integrates product-line safety analysis with model-based development and model checking. It consists of three main procedures :

1) Construct state-based models of a product line having significant, safety-related variations, and explore the relationships between behavioral variations and potential hazardous states through scenario-guided executions of the state model over the variations. 2) Generate properties from product-line requirements for model checking, and create formal behavioral models for each variation. 3) Model check obligations at the variation points to further ensure that those variations upon composition will satisfy the system safety requirements.

In the rest of this chapter, the above mentioned problem and approach will be described in further detail, followed by a discussion of the contributions, and an outline of this dissertation.

1.1 Problem Statement

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"[Bosch (2004), p.256]. In safety-critical product lines such as pacemakers [Ellenbogen and Wood (2005)], mobile communication devices for emergency workers [Yuan and Detlor (2005)], 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, these variations 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 new features and limits the potential for reuse of product-line artifacts. Therefore, creating reusable safety analysis techniques that can incorporate the variations without compromising the safety of individual products is needed.

The development of software product lines currently 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 [Kang (2006)] identifies the following as open problems for the practical use of product lines for safety-critical software:

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

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

3. Accommodating the evolution of the product line and adapting the product-line assets

to the evolved requirements.

The work described in this dissertation addresses these problems in terms of the design, analysis, development and evolution of safety-critical, software product lines.

1.2 A Reusable Safety-analysis Framework

This section gives an outline of the framework presented in this dissertation. This framework takes inputs from the product-line safety requirements, product-line architecture and product-line features, and then performs the following steps (Fig. 1.1):

1. Conduct a product-line safety analysis for each safety-related requirement of the system to create a product-line fault tree [Dehlinger and Lutz (2006)] (used to reason about avoiding the faults). This identifies the components that may contribute to the violation of those requirements, and provides possible causal paths.

2. Create product-line state models [Gomaa (2004); Liu et al. (2005a, 2007c)] for the recorded safety-related components, and derive safety-related scenarios from the fault tree's causal paths. The state models are then simulated against the derived scenarios [Liu et al. (2005a, 2007a,b)]. A scenario execution found to allow illegal/hazardous behavior indicates an unwanted feature interaction. A failure in the execution of a required scenario indicates inconsistencies between the model execution and the specified scenario. In each case, an update to the design is warranted if undesired behavior is detected when exercising the correct scenarios in the state model.

3. Since the state model simulation only tests limited execution paths of the system, to further enhance the safety assurance, this step conducts model checking, an exhaustive search for the undesired scenarios on a given part of the system. To prepare for this, product-line safety requirements are interpreted in terms of temporal logic formulas, [Huth and Ryan (2004)]) and product-line feature behavior models are derived from the product-line features and state models [Liu et al. (2008b)].

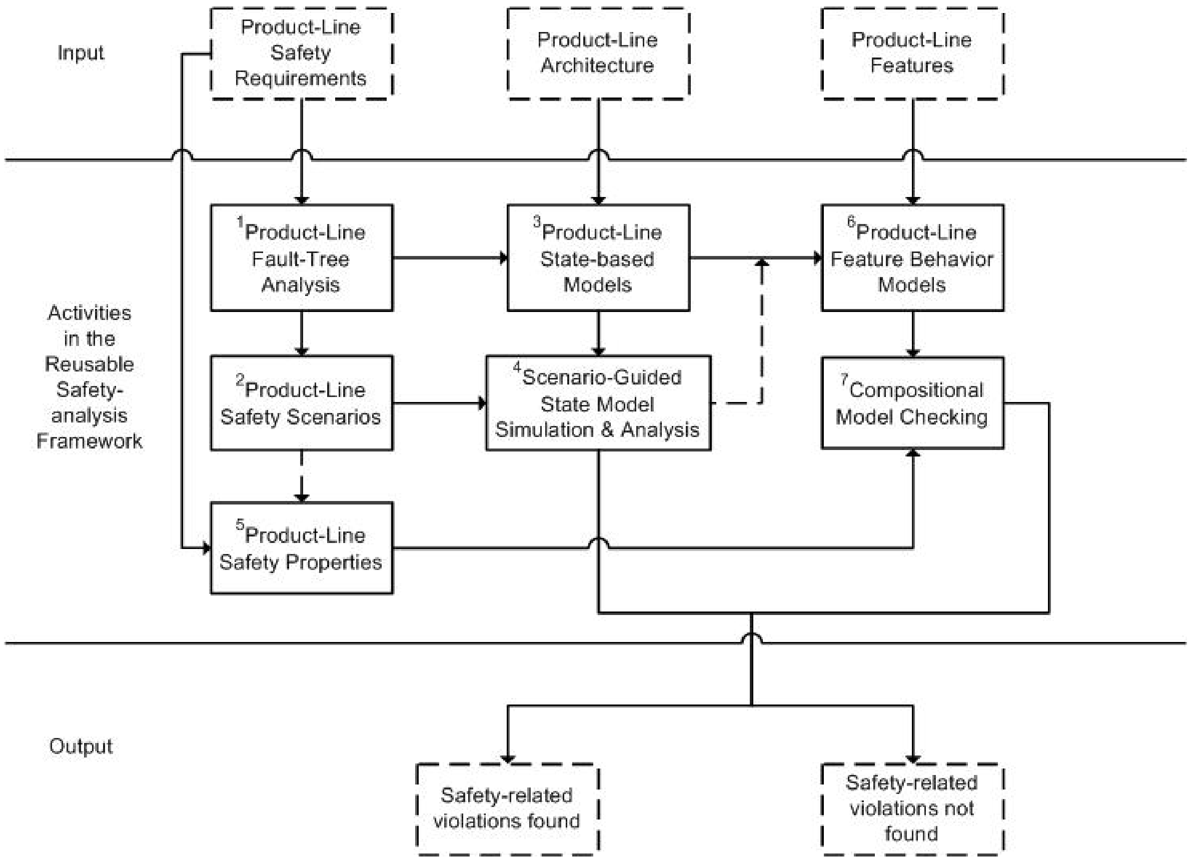

Input
Product-Line Safety Requirements
Product-Line Architecture
Product-Line Features
Activities in the Reusable Safety-analysis Framework
1Product-Line Fault-Tree Analysis
3Product-Line State-based Models
6Product-Line Feature Behavior Models
2Product-Line Safety Scenarios
4Scenario-Guided State Model Simulation & Analysis
7Compositional Model Checking
5Product-Line Safety Properties
Output
Safety-related violations found
Safety-related violations not found

Figure 1.1 A reusable safety-analysis framework

The scenarios and the executions of the state models also provide input to the creation of the properties and feature behavioral models, as they make evident which part in the system relate to a certain safety property. (Their inputs are helpful but not required for performing this step, thus the links between Activity 2 and 5, and Activity 4 and 6 are denoted in dotted lines.)

4. Compositional model checking [Liu et al. (2008a)] is then conducted to generate variation-point obligations at the interfaces between the feature behavior models. This incrementally verifies the satisfaction of the above specified safety properties on the different compositions of features. Confirmation from model checking results further enhances the assurance that different combinations of variations in the product line do not violate those key safety requirements.

Building the product-line state model and feature model promotes reuse among the software product line systems so that a safety requirement can be readily tested and verified with a set of variations.

The outputs of the framework are generated from the scenario execution and the compositional model checking. Any safety-related violations found warrant further investigation of the source of the violations and corresponding updates to the design. The above steps are performed iteratively (i.e., if the output generated from a certain step affects previous steps, those steps need to be repeated) until no safety-related violations are found ([Liu (2007)]).

All the artifacts derived above are reusable assets of the product line. Both the state model and the feature behavior model are revised and re-executed as the product line evolves in response to updates of either the feature diagrams or product-line fault trees.

1.3 Impacts of this Dissertation

The main contribution of this work is that it enables safety-related feature interaction problems to be better investigated and managed. The framework provides four important advantages for safety analysis of product lines. First, 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 models. Second, it allows efficient reuse of model checking results associated with the features in a product line, with no restrictions on how the features can be sequentially composed, thus allowing developers to model check many more real-world systems. Third, it provides tool-supported facilities to guide users in generating, selecting, managing, and reusing product-line safety properties and patterns of properties, which improves the reusability and traceability of formal specifications of product line safety requirements. Fourth, it supports safe evolution of product-line requirements by systematically exploring the impact of changes in the state-based model and accommodating them at the variation points.

1.4 Dissertation Outline

The rest of the dissertation is organized as follows. Chapter 2 presents research background, related work and introduces the running example, a simplified model of a real-world safety-critical product line. Chapter 3 describes the state-based modeling and scenario derivation, as well as the scenario-guided model execution. Chapter 4 presents the support for traceability and manageability regarding the formal interpretation of the safety scenarios (i.e., safety properties) and state-based models (i.e., feature behavior models). Chapter 5 details the incremental and compositional model checking. Support for product-line evolution is discussed in each of the three chapters above, and Chapter 6 offers final conclusions and ideas for future work.

CHAPTER 2. PRELIMINARIES

The work presented in this dissertation is based on the overlapping areas of software product-line engineering, safety analysis, model-based development, and model checking. The focus is on the use of state-based models and compositional model checking strategies to enable verification of safety-related properties in the presence of product-line variations. This chapter reviews the research background information and related work, as well as introducing the running example.

2.1 Background & Related Work

This section surveys the key literature that is related to the work presented in this thesis. It involves topics on software product-line engineering, safety analysis, model-based development, as well as research work on model checking in a product-line setting.

2.1.1 Software Product-Line Engineering

A software product line is a set of software-intensive systems that share a common set of core requirements yet differ amongst each other according to a set of allowable variations [Northrop and Clements (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 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 [Schmid and Verlage (2002)]. 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 [Liu et al. (2007a)]. 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 [Weiss and Lai (1999)]. 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)].

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) [Svahnberg and Bosch (1999)]. 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.

2.1.2 Software Safety Analysis

Safety is defined by Leveson as "freedom from accidents or losses"[Leveson (1995), p.181]. Safety analysis is "a systematic procedure for analyzing systems to identify and evaluate hazards and safety characteristics" [Harms-Ringdahl (2001), p.35]. A hazard is a "state or set of conditions of a system (or an object) that, together with other conditions in the environment of the system (or object), will lead inevitably to an accident (loss event)" [Leveson (1995), p.177].

Safety analysis proceeds in parallel with project development. It helps provide qualitative assurance and identify components that need special care. Moreover, it monitors the effects of changes on system safety. Although safety analysis alone cannot guarantee system safety, it plays a vital role in eliminating and mitigating hazards [Leveson (1995)].

Two of the widely used safety analysis techniques [Leveson (1995); Grunske et al. (2005a)] are:

Fault Tree Analysis (FTA), which conducts a top-down search of faults that can cause hazards and represents such a search in a tree structure. The combination of faults are connected using Boolean logic operators (e.g., AND, OR gates). Each level in the tree structure elaborates the level above it in more detail.

Failure Modes and Effects Analysis (FMEA), which conducts a forward search to determine the effects that individual components' failure modes have on the overall system. The results are recorded in a table.

Safety analysis approaches have also been proposed to verify safety properties and discover missing safety requirements for the multiple systems of a product line. Feng and Lutz [Feng and Lutz (2005)] 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 [Lu and Lutz (2002)]. Yet, these two approaches rely on a static analysis of the product-line requirements rather than the executable analysis done in this work.

More recently, work has been done on 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. An extension of bi-directional, safety analysis approach to product lines [Feng and Lutz (2005)] has been provided 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)]).

DECIMAL is a product-line requirements analysis tool that documents the commonalities, variabilities and dependencies of a product line during the domain engineering phase [Padmanabhan and Lutz (2005)]. 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) [Dehlinger and Lutz (2006)]. 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)]. 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.

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. 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.

2.1.3 Model-Based Software Development

Model-based development of critical systems has demonstrated advantages [Bennett and Rajlich (2000)]. 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)].

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 [Clauß (2001)]; Doerr classifies the relationships within a variation model and relates them to UML notation [Doerr (2002)]; Gomma uses executable UML statecharts as a product-line model [Gomaa (2004)]; and Prehofer uses state-model composition to evaluate the interaction of features [Prehofer (2004)]. The work described in this thesis also uses executable UML but focuses on providing assurance of 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 [Deng et al. (2005)]. Our work concentrates on the impact of software evolution on the safety of the system, rather than on the architectural impact.

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)]. Harel and Marelly, like us, have used a scenario-guided approach to testing state-based models as a way to identify missing requirements [Harel and Marelly (2003)]. 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.

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 work reported here uses executable UML within the Rhapsody software modeling environment as well as the TestConductor tool by Telelogic [Rhapsody (2005)].

2.1.4 Model Checking

Formal verification, the application of rigorous mathematical reasoning to prove that a system satisfies certain properties (or formal specifications) [Rushby (1995)], has gained increasing importance in the software industry. This is particularly true for safety-critical and mission-critical software systems.

Model checking gained favor among various formal verification techniques because of its automated verification procedure and the close resemblance between modeling language and high-level programming languages [Clarke et al. (2000)]. A model checker accepts formal models of system requirements and design [Havelund et al. (2001); McMillan (1993)] or sometimes implementation [Corbett et al. (2000)], as well as some desired or undesired properties of the system [Clarke et al. (2000)], and then employs systematic exploration of the execution paths (exhaustive exploration if the model size is manageable) of the model to see if those properties are satisfied or not [Huth and Ryan (2004)]. Researchers have used model checking to find flaws in software designs that are otherwise hard to detect [Havelund et al. (2001)], e.g., by testing.

If a single product can be decomposed into a set of units that can be designed and implemented separately, we call them "individually-behaving units" (e.g., components in component-based system [Atkinson et al. (2002)] or collaborations in collaboration-based protocol design [Geppert and Rößler (2004)]). A product line can also be viewed as different compositions of such units [Northrop and McGregor (2003)]. No matter which form the product line is taking, one obstacle to model checking product lines is reuse management that accurately reflects the variations. The commonalities (or common units) in product lines make it possible to reuse some model checking effort (e.g., models created, properties specified). However, the variations (or different units) and the new dependencies (constraints between commonalities and variations, or among variations) or different composition of units they introduce can make it difficult to identify the properties to verify for each variation introduced. This currently limits the potential for reuse of model-checking assets.

Thus, there is a need for both: 1) property specification techniques that can incorporate the concept of product-line variation and reuse; and 2) techniques that can effectively use compositional verification in a product line setting.

For the first need, existing work has indicated the possibility of successfully conducting model checking for software product lines, e.g., Kishi and Noda [Kishi and Noda (2006)] proposed an approach that models product-line variations in UML models and then translated them into SPIN models; Li, Krishnamurthi, and Fisler [Li et al. (2005)] have exploited compositional verification in the product-line context by automatically checking interfaces of separate features using the labeling algorithm in CTL model checking; and Robby, Dywer, and Hatcliff [Robby et al. (2006)] have constructed Bogor, an extensible model-checking framework that can be customized to tailor to different application domains, e.g., to be used as a back-end model checker for Cadena [Childs et al. (2006)]. Cadena is an integrated environment for building and modeling CORBA Component Model systems that can be used to develop model-driven component-based product lines.

A major issue remaining unaddressed in work to date is the management of property specifications at the product-line scope. Traditionally, the properties being verified are derived from requirements [Konrad and Cheng (2005)] or subsystem/component/interface specifications [Kurshan (2004)]. Several techniques have been developed to ease the difficulty of translating informal (natural language) specifications into formal ones (e.g., temporal logic formulas [Huth and Ryan (2004)]), such as the Property Specification Patterns [Dwyer et al. (1999)] and various work that helps select and adapt those patterns [Corbett et al. (2000); Jörges et al. (2006); Loer and Harrison (2006); Mondragon et al. (2003); Smith et al. (2002)], a set of tools to help edit the LTL temporal logic properties in a communication diagrams [Autili et al. (2006); Smith et al. (2001)], techniques that translate a subset of natural language [Holt (1999); Konrad and Cheng (2005)] or specification language (syntactic sugar) [Beer et al. (2001)] into temporal logics, and syntax-directed editing environment [Reps and Teitelbaum (1984)]. However, these techniques, to the best of our knowledge, do not treat property specification in a reusable setting.

For the second need, our work falls in the context of open-system verification where features are added in a sequential fashion. Blundell, Fisler, Krishnamurthi and Hentenryck [Blundell et al. (2004)] propose a framework in which interface obligations are generated as temporal properties. Our technique differs from theirs in that [Blundell et al. (2004)] requires interface states (here, the variation points) to be terminal states with no outgoing transitions, while our approach does not have this restriction. Secondly, we permit features to be added in ways that allow intra and inter-feature loops, a flexibility that was needed to accurately model the pacemaker product line. Wang [Wang (2005)] extends [Blundell et al. (2004)] and allows inter-feature loops. However, Wang (2005) assumes that interface states are sufficient for composing different features and does not re-explore the non-interface states. This implicitly puts a restriction on the type of inter-feature loops that can be verified.

In [Thang (2005)], Thang presents the necessary conditions which, when satisfied by the base and its extension feature(s), ensure that the property verification results hold before and after the base is extended with the corresponding features. Though the work allows loops between the base and the extensions, it does not provide insights into the cases where the necessary conditions are violated.

Xie and Browne [Xie and Browne (2003)] model each component (e.g., of a product line) as an Asynchronous Interleaving Message-passing (AIM) computation model (i.e., only one model can execute at a time), and use assume-guarantee to generate assumption on the environment (i.e., other components in the composition) for a given component. The interface between two components is the input and output message types between the two. The system verification is carried out on the abstraction of the composed model, which is obtained from the environmental assumptions, the verified properties, and the messaging interfaces of the constituting components. Their work differs from ours in that both their computation model (especially the type of interface) and model checking strategy are suitable for verifying the communication between components, while ours targets the increments of functionality of one or more systems.

Our work falls into the category of compositional verification [Abadi and Lamport (1995)]. We use sequential composition (Def. 2) rather than parallel composition, as in, e.g., [Giannakopoulou et al. (2002); Basu and Ramakrishnan (2006)], because it would add unnecessary complexity to the state space and obscure the interfaces among features that we want to maintain in a product-line setting for effective reuse.

2.2 Running Example

To illustrate the process outlined in Chapter 1, we use a pacemaker product line as a running example throughout the rest of the dissertation. 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. Its major functions include detecting abnormal cardiac rhythms (including tachycardia and bradycardia, which are fast and slow abnormal heart beats, respectively), and applying therapies (e.g., stimulating the heart with an electrical pulse or shock). The therapies are applied to two chambers of the heart: ventricle and atrium [Ellenbogen and Wood (2005)]. It is safety-critical because some failures can damage the patient's health or even lead to loss of life [Ellenbogen and Wood (2005); Littlewood and Strigini (1993)].

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 (2005)]. 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 running example considers four different products within the pacemaker product line: BasePacemaker, RateResponsivePacemaker and ModeTransitivePacemaker and ModeTransitive-RateResponsivePacemaker. Figure 2.1 shows four products in the pacemaker product line [Liu et al. (2007a,b)]:

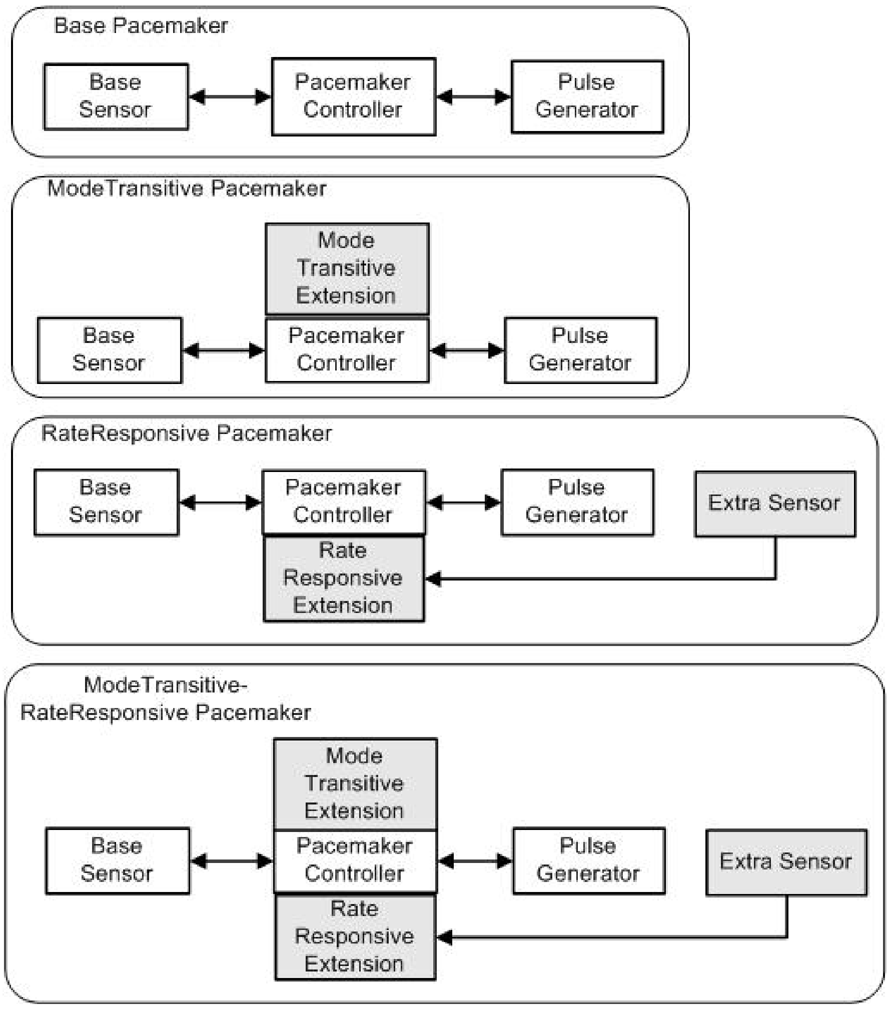


Figure 2.1 Pacemaker product line overview

BasePacemaker has the basic functionality shared by all pacemakers: generating a pulse if no heart beat is detected during the sensing interval. This mode of execution is called Inhibited Mode.

ModeTransitivePacemaker has an additional feature called ModeTransition Extension that enables it to switch between Inhibited Mode and TriggeredMode during execution. In the TriggeredMode, a pulse follows every heartbeat to regulate the heartbeat.

RateResponsivePacemaker has an additional sensor, Extra Sensor, that can detect a patient's activity level (i.e., respiration rate while resting vs. while exercising) and adjusts the sensing interval (to normal or upperRateLimit) accordingly. 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.

ModeTransitive-RateResponsivePacemaker combines the features of the ModeTransitivePacemaker and the RateResponsivePacemaker to provide both inhibited and triggered heartbeat regulation and adaptation to patient's activity level.

A **safety property** motivates and explains the method: *In the InhibitedMode, the pacemaker shall always generate a pulse when no heartbeat is detected during the normal sensing interval.* 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.

Previous work has been done in state-based modeling of a pacemaker by Goseva-Popstojanova et al. [Goseva-Popstojanova et al. (2003)], by Douglass [Douglass (1999)] and as examples with the Rhapsody toolset from Telelogic. 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.

CHAPTER 3. SAFETY-ANALYSIS GUIDED STATE-BASED MODELING

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[1] 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.

3.1 Introduction

The state-based modeling approach in this chapter 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. (2007b)]:

1. Analysis and modeling of timing/ordering-sensitive failure events to determine their possible safety implications.
2. Simulation of the behaviors described by the requirements in the fault tree, to illustrate the violation of a safety property.

[1]The work presented in this chapter is adapted from Liu, J., Dehlinger, J., and Lutz, R. (2007a). Safety analysis of software product lines using state-based modeling. *Journal of Systems and Software*, 80(11):1879-1892.

3. 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 chapter 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 chapter 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 rest of the chapter is organized as follows. Section 3.2 gives an overview of the approach and introduces the running example (a pacemaker product line). Section 3.3 describes the state-based modeling and execution of the product line to validate safety properties. Section 3.4 discusses and evaluates the results in terms of their use in safety-critical systems. Finally, Section 3.5 shows how this approach can be used to support the safe integration of new features into the product line.

3.2 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 3.3, where they are applied to the pacemaker product line.

3.2.1 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.

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).

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); Padmanabhan and Lutz (2005)]. 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 first 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); Padmanabhan and Lutz (2005)] for details.

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.

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.

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.

3.2.2 Software Tools and Application

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 Telelogic.

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 statecharts 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 statecharts 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 3.4 how this affected the validation process.

To illustrate the process outlined in Section 3.2.1, we use the pacemaker product line introduced in Section 2.2 as a running example throughout Sections 4 and 5.

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

Table 3.1 Products and their variations

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

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.

3.3 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.

3.3.1 Commonality and variability analysis

Requirements and features for a product line are often specified in a Commonality and Variability Analysis (CVA) [Ardis and Cuka (1999); 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 Figure 3.1. The variations distributed among product line members are shown in Table 3.1.

3.3.2 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.

Figure 3.1 Excerpts from pacemaker product-line Commonality and Variability Analysis

Step1: 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.*

Step2: 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 Figure 3.2 shows an example in the bottom left node “No pulse generated by the end of 300 msec sensing time”. Each leaf node within this fault tree refers to either one of the commonalities or variations described in Section 3.3.1 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.

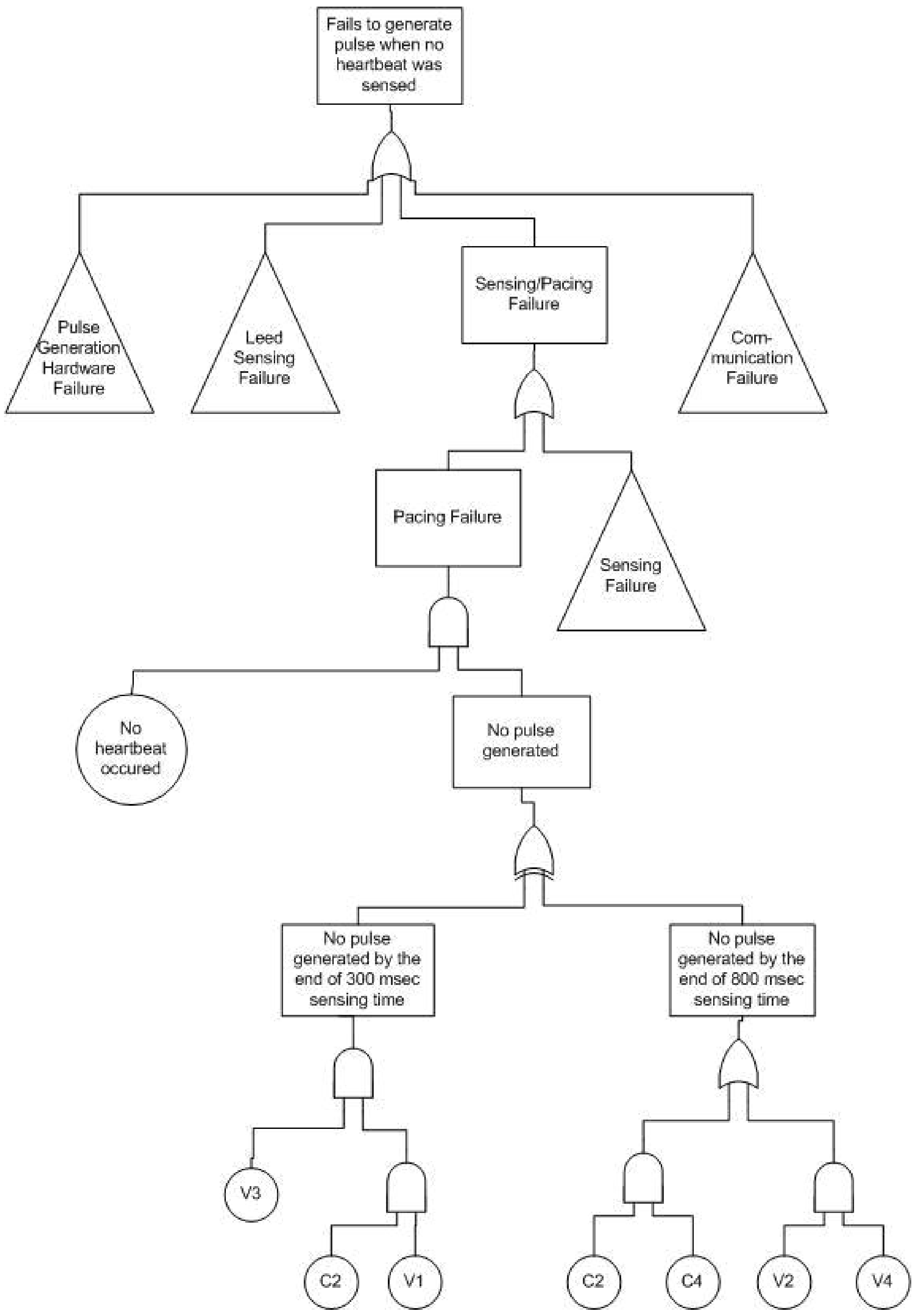

Fails to generate pulse when no heartbeat was sensed
Pulse Generation Hardware Failure
Leed Sensing Failure
Sensing/Pacing Failure
Com-munication Failure
Pacing Failure
Sensing Failure
No heartbeat occured
No pulse generated
No pulse generated by the end of 300 msec sensing time
No pulse generated by the end of 800 msec sensing time
V3
C2
V1
C2
C4
V2
V4

Figure 3.2 Excerpt of pacemaker PL-SFTA

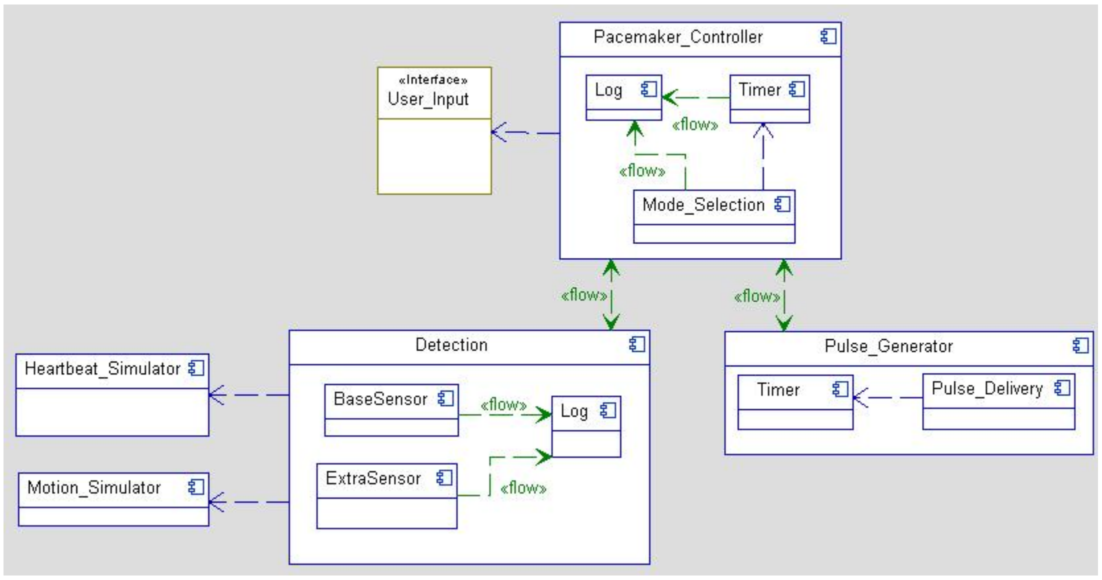


Figure 3.3 Pacemaker architectural configuration

3.3.3 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 chapter 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 (2004); Bosch (2000)]. Fig. 3.3 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. (2005b,a)].

Step1: 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. 3.2 is generated by, and thus here associated with, the component "Heartbeat Simulator". Similarly, the leaf node V3 from Fig. 3.2 (that allows extra sensors) is tied

Table 3.2 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

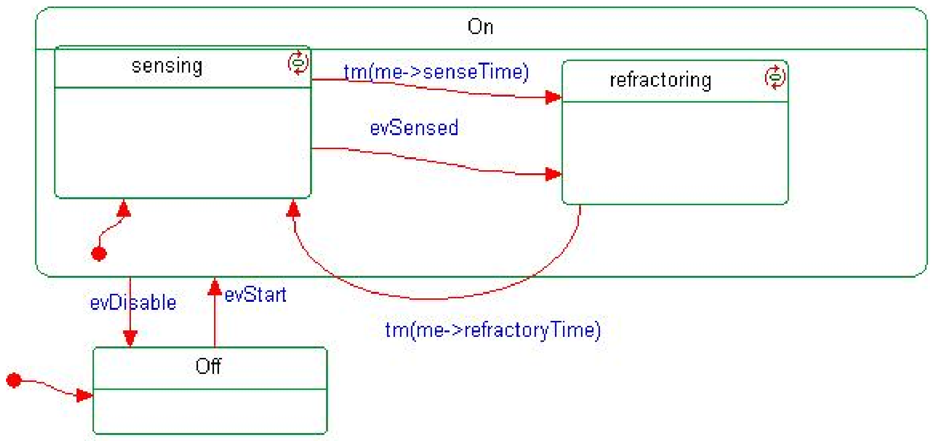


Figure 3.4 Pacemaker Controller in BasePacemaker

to the component ”ExtraSensor” in Fig. 3.3. Table 3.2 describes the mapping between the components in Fig. 3.3 and the leaf nodes in Fig. 3.2. (Note that C1 was not shown in Fig. 3.2 for readability.)

Step2: 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. 3.4), 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.

1. 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. 3.4, 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 "Refractoring" sub-state. This statecharts displays the behavior that is common to all the pacemaker products in the product line.

2. Adding RateResponsivePacemaker functionalities. The RateResponsivePacemaker's statecharts 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 statecharts. 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 statecharts in Fig. 3.4 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".

3. Adding ModeTransitivePacemaker functionalities. The statecharts 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 statecharts in Fig. 3.4 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 statecharts 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 (2004)]. When a new variation is introduced, its corresponding statecharts inherits the existing statecharts and uses condition connectors (whenever necessary) to model dependencies between different variations.

Since our goal is safety analysis, the statecharts 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 statecharts 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 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 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 statecharts.

Step3: 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 BasePacemaker), 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 statecharts model so this binding refers to whether or not to include a certain piece in the code-generation. For example, the Product-Line statecharts in Fig. 7 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 section.

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.

3.3.4 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.

Step1: 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. 3.5 models an initial forbidden scenario for the root node of the fault tree shown in Fig. 3.2, "Excerpt of Pacemaker PL-SFTA", as a sequence diagram. The root

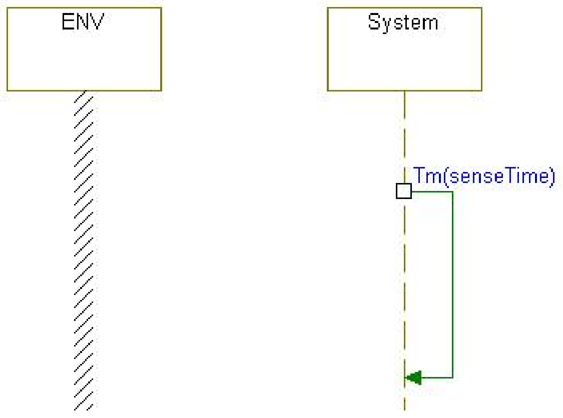


Figure 3.5 Initially derived scenario

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. 3.5 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.

Step2: 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 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. 3.6.

The sequence diagram in Fig. 3.6 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. 3.6 depicts a scenario in which

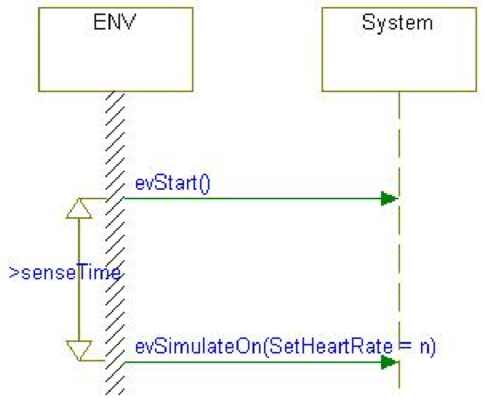


Figure 3.6   Refined scenario

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 3.3.3) 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.

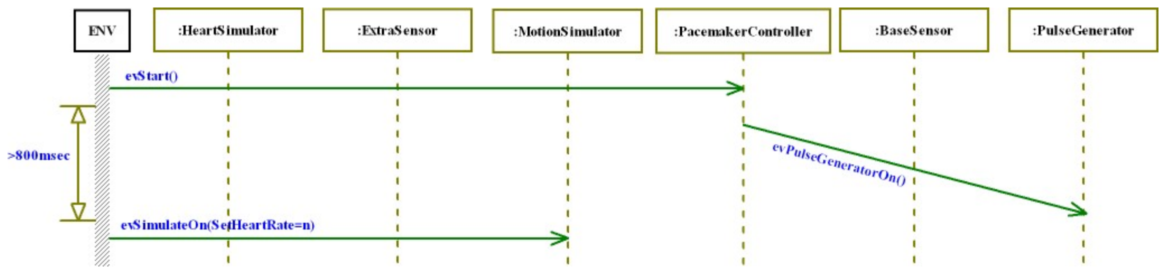


Figure 3.7 Scenario derived from PL-SFTA

3.3.5 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.

Step1: 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. 3.7 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).

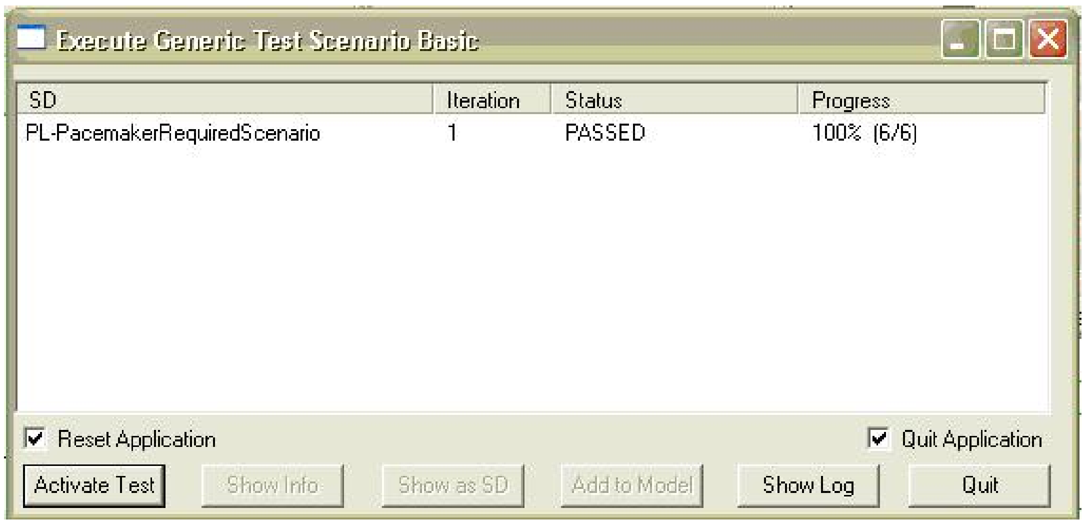
The generic sequence diagram must also include the external events representing messages generated from the system border of the sequence diagram, e.g. the evStart() event in Fig. 3.7, and internal events that occur between the internal instances of the sequence diagram, e.g., the evPulseGeneratorOn() event in Fig. 3.7. Variations of data associated with the events, if possible, are also specified here. For example, in Fig. 3.7, “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.

Step2: Verify the state model

Each customized state model is now executed against its corresponding scenarios. 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, 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


Execute Generic Test Scenario Basic

SD	Iteration	Status	Progress
PL-PacemakerRequiredScenario	1	PASSED	100% (6/6)

☑ Reset Application ☑ Quit Application

Figure 3.8 Output from TestConductor

named scenario in the right order. Fig. 3.8 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 3.4.1.

3.4 Discussion of Results

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

3.4.1 Applying the Results to Enhance Safety

There are two main uses of the results to enhance the safety of the product line in this application: finding design errors, and addressing safety-related concerns.

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 3.3.1. The updating of the CVA with new product-line requirements may then require additions/modifications to the product line's hazard analyses, described in Section 3.3.2 to account for the new possible hazards introduced by the new requirements. Following this, an additional iteration of Steps 3-5, described in Sections 3.3.3 - 3.3.5, would be necessary to assess the safety properties of the product line given the new product-line requirements.

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 and validates any mitigation steps. For example, from the PL-SFTA it was 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. 3.9 shows that the sensing interval is required to be less than 300 msec 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 msec due to the failed extra sensor. This demonstrates that V3 is indeed a credible potential vulnerability that

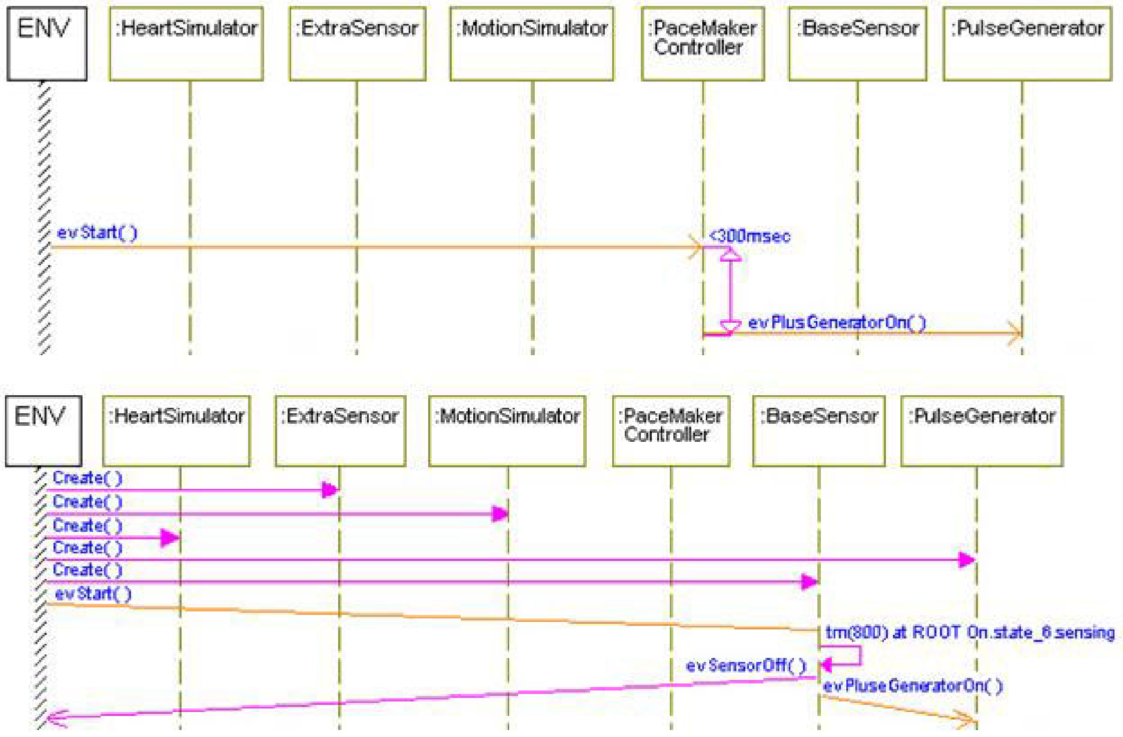


Figure 3.9 Comparison between the required scenario (upper part) and the actual scenario (lower part) when V3 is present

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 3.3.2 - 3.3.5 to validate the product line's safety properties with the new requirements.

3.4.2 Limitations

Although Rhapsody's executable state models support real-time notions, we found that it cannot enforce exact real-time measurement (e.g., border time values) 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. 3.8) 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 Step 2 of Section 3.3.4), some generality is lost in depicting the testing scenario. This seems to be an unavoidable problem at the moment but future tool development may ease this difficulty.

3.5 Support for Evolution

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.

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) [Svahnberg and Bosch (1999)].

This section[2] describes how the tool-supported state-based safety analysis approach presented in this chapter can be used to determine if new features can be safely integrated into the

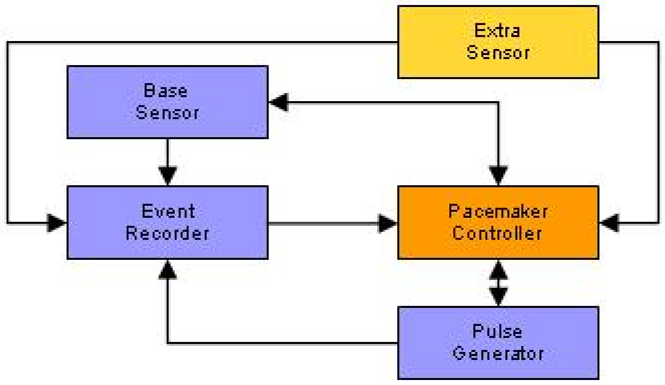


Figure 3.10 Product-Line architecture after evolution

product line without introducing unchecked safety concerns. The potential feature interactions that need to be modeled are scoped and identified with the aid of product-line software fault tree analysis. Further, reuse of the state-based models is effectively exploited in the evolution phase of product-line engineering.

3.5.1 Illustrative Example

To illustrate our approach, we build upon the pacemaker product line described in Section 2.2. The evolution of the pacemaker product line that we consider here involves the addition of an EventRecorder component, shown in Fig. 3.10, 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 3.3.

The addition of the EventRecorder feature was included into the product line's requirements using DECIMAL [Padmanabhan and Lutz (2005)].

[2]The work presented in this section is adapted from Liu, J., Dehlinger, J., Sun, H., and Lutz, R. (2007b). State-based Modeling to Support the Evolution and Maintenance of Safety-Critical Software Product Lines. In *ECBS 07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 596-608, Washington, DC, USA. IEEE Computer Society.

Table 3.3 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

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.

3.5.2 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. 3.11, 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

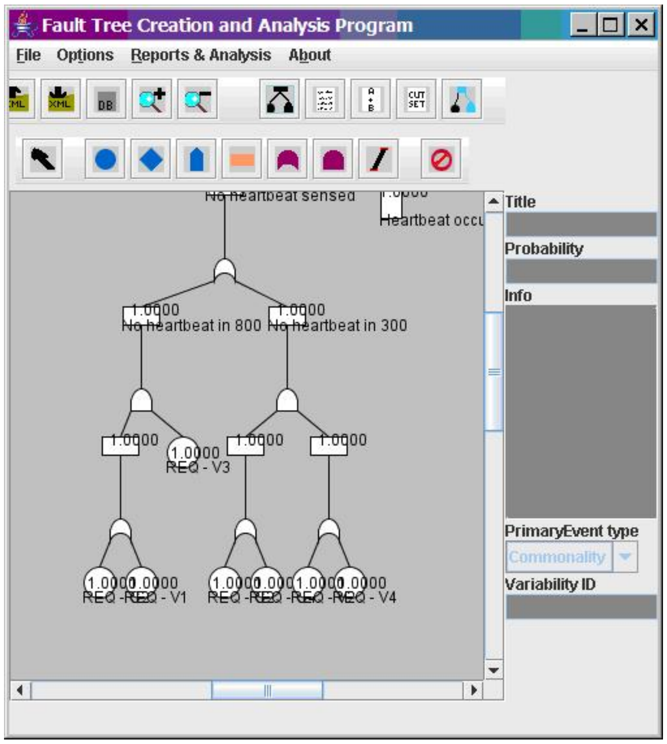


Figure 3.11 Excerpt of pacemaker product-line SFTA in PLFaultCAT after evolution

as well as those features from existing product-line products. For example, as illustrated in Fig. 3.12, the Mode-Transitive 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 3.5.3, to those feature interactions, like shown in Fig. 3.12, which may need to be further scrutinized.

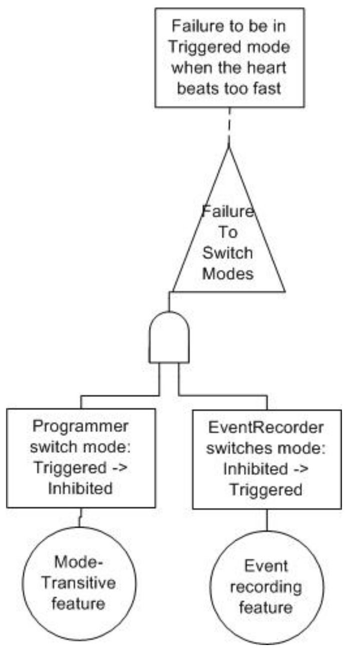

Failure to be in Triggered mode when the heart beats too fast
Failure To Switch Modes
Programmer switch mode: Triggered -> Inhibited
EventRecorder switches mode: Inhibited -> Triggered
Mode-Transitive feature
Event recording feature

Figure 3.12 Excerpt of product-line SFTA illustrating potential feature interactions

3.5.3 State-Based Modeling Evolution

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. 3.12. 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. 3.12, 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 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 3.3.5, 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. 3.13 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 statecharts in Fig. 3.14 (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. 3.12, 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,

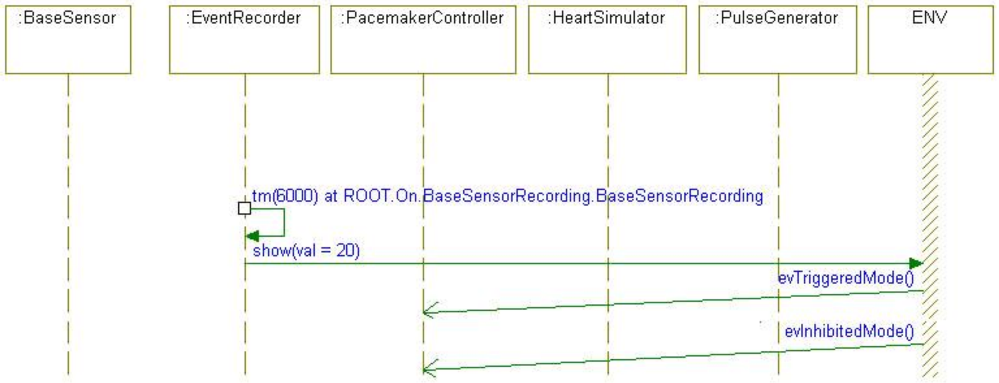


Figure 3.13 Excerpt of pacemaker animated sequence diagram

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

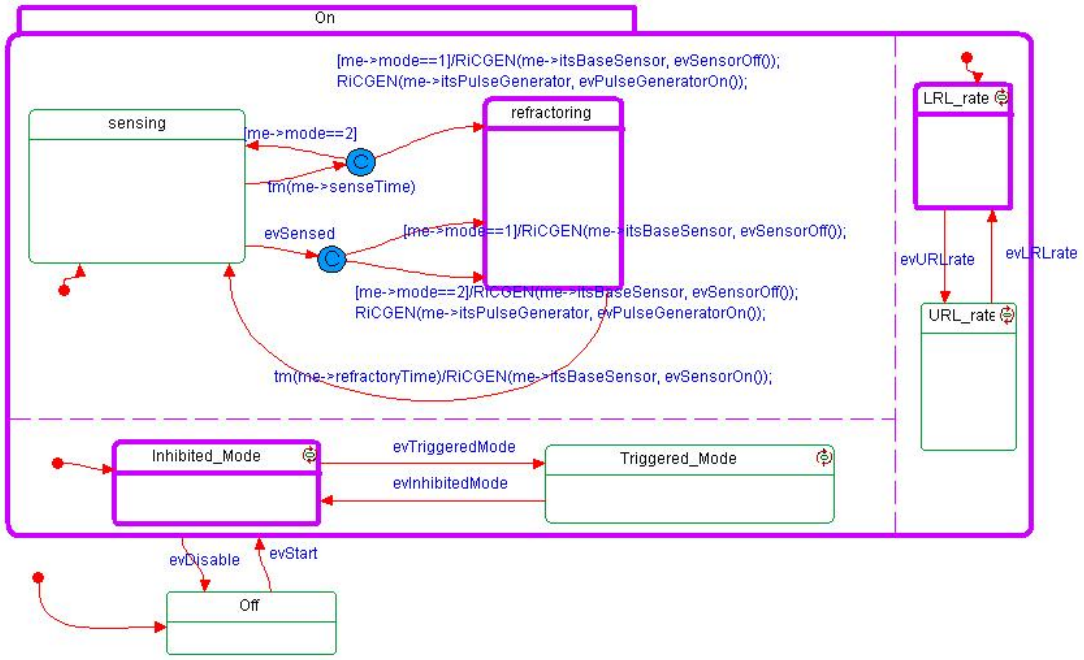

On
[me->mode==1]/RiCGEN(me->itsBaseSensor, evSensorOff());
RiCGEN(me->itsPulseGenerator, evPulseGeneratorOn());
sensing
refractoring
[me->mode==2]
tm(me->senseTime)
evSensed
[me->mode==1]/RiCGEN(me->itsBaseSensor, evSensorOff());
[me->mode==2]/RiCGEN(me->itsBaseSensor, evSensorOff());
RiCGEN(me->itsPulseGenerator, evPulseGeneratorOn());
tm(me->refractoryTime)/RiCGEN(me->itsBaseSensor, evSensorOn());
LRL_rate
URL_rate
evURLrate
evLRLrate
Inhibited_Mode
Triggered_Mode
evTriggeredMode
evInhibitedMode
evDisable
evStart
Off

Figure 3.14 Animated PacemakerController statecharts in the evolved product-Line state model

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.

3.5.4 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 [Clauß (2001)]. 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 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.

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 [Padmanabhan and Lutz (2005)], the product-line requirements documentation and analysis tool, coupled with the use of PLFaultCAT [Dehlinger and Lutz (2006)], 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.

CHAPTER 4. FORMAL SPECIFICATION OF SAFETY REQUIREMENTS

Property specification in model checking is currently handled without adequately taking software product lines into account. This is largely due to the fact that the available model checkers and property specification tools lack sufficient support for reusing model-checking effort. The challenge is twofold: first, we need to make the properties accurately trace to individual system requirements and models even as they evolve; and second, we need to make the property specification easy to share and reuse among different systems of the same product line. The contribution of this work[1] is a tool-supported technique to guide users in generating, selecting, managing, and reusing product-line properties and patterns of properties. The technique is evaluated in a product-line application. Results show that it improves the reusability and traceability of property specifications for model checking in a product line setting.

The original version of the FormulaEditor Tool was developed at Avaya Labs Research, NJ, USA, as part of the SELEX project.

4.1 Introduction

Model checking is a powerful technique for enhancing the quality of software systems [Clarke et al. (2000)], e.g., by identifying flaws that would not have been caught otherwise [Havelund et al. (2001); Kaivola (2005)]. However, there is currently insufficient support for model checking in product lines, most specifically, for property specification and management.

In this chapter we present a detailed technique, supported by a property specification and management tool, FormulaEditor, for helping with the model checking of software product

[1]The work presented in this chapter is adapted from Liu, J., Hauptman, M., Lutz, R., Geppert, B., and Rößler, F. (2008b). A tool-supported technique for specification & management of model-checking properties for software product lines. Technical Report, 08-05, Computer Science, Iowa State University.

lines. The core of the technique is a product-line-oriented user interface to guide users in generating, selecting, managing, and reusing useful product line properties, and patterns of properties for model checking. The tool also associates the properties with the requirements, models and verification results of each product in the product line so that any changes can be readily traced and the properties updated accordingly. The technique is evaluated in applications to telecommunication-protocol and cardiac-pacemaker product lines.

In a product line, some requirements, called *commonalities*, are shared by all the products. For example, a commonality for the pacemaker product line described below is, "When a heartbeat is detected during the SenseTime, the pacemaker shall not generate a pulse." This property is to keep the pacemaker from giving a pulse when it is unneeded by the patient. The difficulty with reuse across a product line is that the differences among the products, called their variations, can complicate the implementation and verification of the properties. For example, some pacemakers can distinguish whether a patient is exercising or at rest, and adjust the SenseTime accordingly. This variation means that the specification of the common property described above, in fact, varies slightly among products. Verification that each new system built in a product line satisfies the common properties takes many forms including inspection, state-based simulation, and testing [Atkinson et al. (2002); Liu et al. (2007a); Weiss and Lai (1999)]. However, these techniques do not provide the coverage or level of assurance needed for products in some domains. For example, in our work with both communication protocols and pacemakers, we found that more rigorous verification to show key properties held in each new product was needed. Model checking, in particular, can provide insights into rare and boundary cases. We thus wanted to be able to model check these properties in the product lines.

Product-line verification, like product-line engineering in general, tries to reuse whatever is common across the product line to reduce the cost and increase the quality of each new product. However, correctly specifying and keeping track of the properties across a product line is a challenge. This is especially true in large product lines where new features are added regularly in response to market pressures. The many commonalities in a product line urge reuse, but the variations among the products demand very careful management of that reuse.

The challenge for model checking a product line is twofold. First, we need to make properties precisely and accurately trace to individual system requirements and models even as they evolve. Second, we need to make the property specifications easy to share and reuse among different systems of the same product line. For example, how do we know if a property should still be satisfied in the presence of some specific variations? For a new system in the product line, how can we easily reuse properties from other product-line members? In this chapter we present a technique that is able to associate properties with variations without compromising the completeness or accuracy of properties for individual products. The property specifications captured and maintained in the tool thus become reusable assets of the product line. Thus, verification of common patterns can be enforced in all products in the product line. By making it easier to specify and manage the properties, we hope to extend the use of model checking in product lines. Moreover, this approach and tool can benefit not only safety-critical product lines (e.g., pacemakers, mobile communication devices for emergency workers, constellations of satellites, and medical-imaging systems), but also single systems that experience frequent maintenance or evolution.

The rest of this chapter is organized as follows. Section 4.2 describes the method by presenting a motivating example followed by design rationale and tool support. Section 4.3 evaluates our technique in a pacemaker product line case study and discusses the results.

4.2 Method

This section describes our property specification technique for software product lines, including a motivating example, the design rationale behind the work, and a description of the tool support.

4.2.1 Motivating Example

The work was motivated by the need to model check a family of communication protocols that resulted from an Avaya refactoring project [Geppert et al. (2005); Geppert and Rößler

(2004)]. In the following, we call the members of the protocol family "protocol variants". Each protocol variant is composed of a set of smaller building blocks (called collaborations) that encapsulate behavior across agent boundaries. Agents are the distributed participants in the communication the protocol regulates Common collaboration examples are connection establishment, connection tear down, and authentication.

There are two mechanisms for composing collaborations. The first mechanism is message multiplexing. If we treat each collaboration as a micro-protocol (in contrast to the composite protocol after composition), the outbound messages sent by the micro-protocols (we call them "micro-messages") must be multiplexed together to form a composite message and be considered as one unit for transportation over the network. Consequently, any incoming composite messages need to be demultiplexed into micro-messages to be handled by micro-protocols upon arrival.

The second mechanism is sequencing, which manages the causal dependencies among the collaborations. We can divide a protocol agent into several roles, one for each collaboration in which the agent participates. The roles represent independent state machines. Protocol sequencing takes care that the roles are executed in the right order. In contrast to traditional protocol design where a protocol is viewed as the composition of protocol agents (each of which can be represented by a state machine), the collaboration-based design enables users to analyze, test, and change cross-cutting behavior independently, thus allowing easier evolution and maintenance [Geppert et al. (2005); Geppert and Rößler (2004)].

Figure 4.1 shows part of a protocol for registering IP phones at IP telephony servers. There are four agents, namely the *endpoint*, the *station server*, the *gatekeeper*, and the *environment*. (The *environment* is not shown in Fig. 1 but is needed for modeling user input and making the protocol state machine finite). The *environment* and *gatekeeper* together with the *endpoint* implement the *authentication* collaboration. It is a four-way handshake to authenticate an *endpoint*. The *environment*, *endpoint*, *gatekeeper*, and the *station server* collaborate to implement the *associate-station* collaboration. It consists of one two-way handshake between each of the two adjacent agents in order to allocate necessary resources for an *endpoint*.

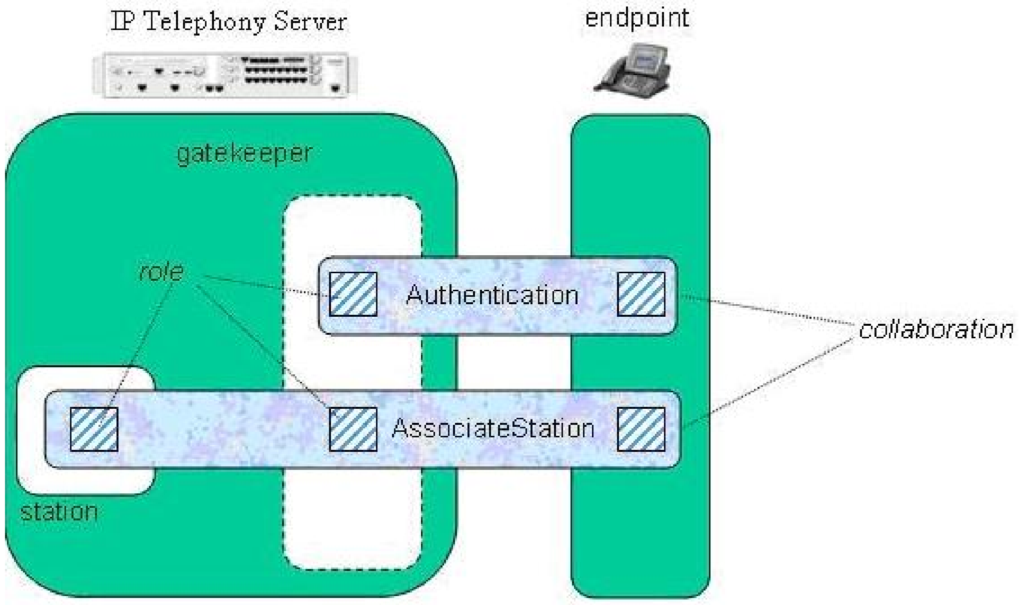


Figure 4.1 Registration protocol (partial overview) [Geppert et al. (2005)]

As a motivating example, a simplified protocol product line consists of three protocol variants: 1) the authentication protocol includes the authentication collaboration only, 2) the associate-station protocol includes the associate-station collaboration only, and 3) the authentication-associate-station protocol includes the composition of the two collaborations. The complete protocol product line is much more complex due to a larger number of collaborations and compositions.

An example property (specified as a natural language description) for the authentication protocol is: "It is always the case that whenever the environment receives a REGISTER_CONFIRM message, the authentication role in the gatekeeper is in authenticated state". An example property for the associate-station protocol is "It is always the case that whenever the environment receives a REGISTER_CONFIRM message, the associate-station role in the gatekeeper is in bound state and the associate-station role in the station server is in stim state".

The motivation for this tool is the recognition that the properties for these two protocols have a lot in common. For example, the example properties in both protocols share the following parameterized property: It is always the case that whenever the environment receives a REGISTER_CONFIRM message, the A role in the B agent is in C state. A, B, C are parameters to be instantiated by concrete role, agent, or state in that specific protocol. A similar pattern can apply to the other properties between the two protocols. This led us to provide parameterized properties that could be reused for different collaborations in different protocol variants.

Since the authentication-associate-station protocol is the composition of the above two collaborations, not only properties from the two collaborations need to be verified again in this protocol, but also the properties regarding the compositional logic need to be verified. This includes making sure that the compositional mechanism is correctly enforced as well as that the composed protocol is behaving as expected. An example of the former is that any incoming composite message is always completely consumed, meaning that all of its inbound micro-messages are always eventually consumed. An example of the latter is that in the authentication-associate-station protocol, the authentication collaboration has to be successful before the associate-station collaboration can be invoked.

An example property for the authentication-associate-station protocol is: “It is always the case that whenever the environment receives a REGISTER_CONFIRM message, the Associate-Station role in the gatekeeper is in bound state, the Authentication role in the gatekeeper is in authenticated state, and Associate-Station role in the station server is in stim state”.

Due to space limitations we cannot list all the properties of the authentication-associate-station protocol here. In fact, if more than one collaboration is involved in a protocol, the number of relationships among incoming/outgoing messages, inbound/outbound micro-messages, and the various states of different roles in the protocol agents, can be exponential.

The key to addressing this problem is to provide tool support to allow recurring patterns in the product line to be identified and instantiated so that a batch of properties can be created and reused. The use of recurring patterns not only contributes to the completeness of the properties, but also speeds up specification.

The potential patterns include but are not limited to the following two categories: patterns of collaborations and patterns of compositional logic. An example of the former is that “It is always the case that whenever the environment receives a REGISTER_CONFIRM message, the A role in the B agent is in C state". An example of the latter is that "any incoming composite message is always completely consumed". Those patterns, once identified, became an asset of the product line so that verification of common patterns can be enforced in all products.

4.2.2 Design Rationale

A straightforward way to check the properties for each of the three protocol variants of the protocol product line described above is to specify each of them one by one and invoke a model checker to verify them. However, for even a small-scale product line, e.g., of twenty collaborations and ten possible compositions, the work involved in tracking all the properties to be verified for each protocol variant became quite hard to manage. Gearing the product-line property specification process to reuse is therefore of great importance to model-checking industrial product lines.

We identified the following needs for property specification and management in a product-line scope. These needs provided the design rationale behind the tool we developed:

1. The tool needs to keep track of which requirement(s) each property specification is derived from and to make use of common property patterns in the product line. This includes: 1) associating properties with individual requirements. Note that the satisfaction of one requirement may entail verification of several properties, each targeting a different aspect of the requirement, e.g., requirement on composition mechanism can turn out to be properties regarding all the micro-messages being consumed correctly; and 2) introducing product-line specific property patterns that can be instantiated for individual members of the product line. In contrast to the Property Specification Patterns [Dwyer et al. (1999)], the product-line specific patterns are intended as reusable assets for the specific product line only. It is thus possible that two different product-line patterns share the same Property Specification Pattern while differing in the instantiation rules for their parameters (e.g., some parameters may only be able to be instantiated by messages from a certain agent) or their scope (e.g.,

some pattern can only be used in certain protocol variants).

2. The tool needs to keep track of which product-line member it is targeting. This includes associating properties with their models, as well as with the verification results. This is because the models determine the validity of the properties being specified. If a property is shown to be false, the requirement may need to be modified (e.g., such a property may be an explanation for an ambiguous requirement).

The next section will describe how the above design rationale is supported by our tool.

4.2.3 Tool Support

The work presented in this section demonstrates a model checking management tool, FormulaEditor, which we developed to support the property specification and management for product lines.

The architecture overview of the tool is presented in Figure 4.2. The figure shows the tool being applied to a product line of three members A, B, and C. The part enclosed in dotted lines is not part of the tool but serves as input to it.

The tool has three main functions: project management, property editing and model checking.

Project management. As mentioned before, property specification is not an isolated process. Therefore, the tool helps users manage the resources needed for linking the property specification with the rest of the model checking and product line development effort.

Each product line is managed as a project. The tool provides a project configuration interface to let users create and specify settings for resources that may be shared by the entire product line (e.g., model checker location, common property file etc.). Those common settings are loaded every time the project is opened, e.g., the common-property pattern file will show up in property-editing for all models in the project.

A property table contains all the properties specified for that product (e.g., see Figure 4.5). The importing-property-table and import-project facility in the project management interface allow a set of properties specified for one system or one product line to be copied to another

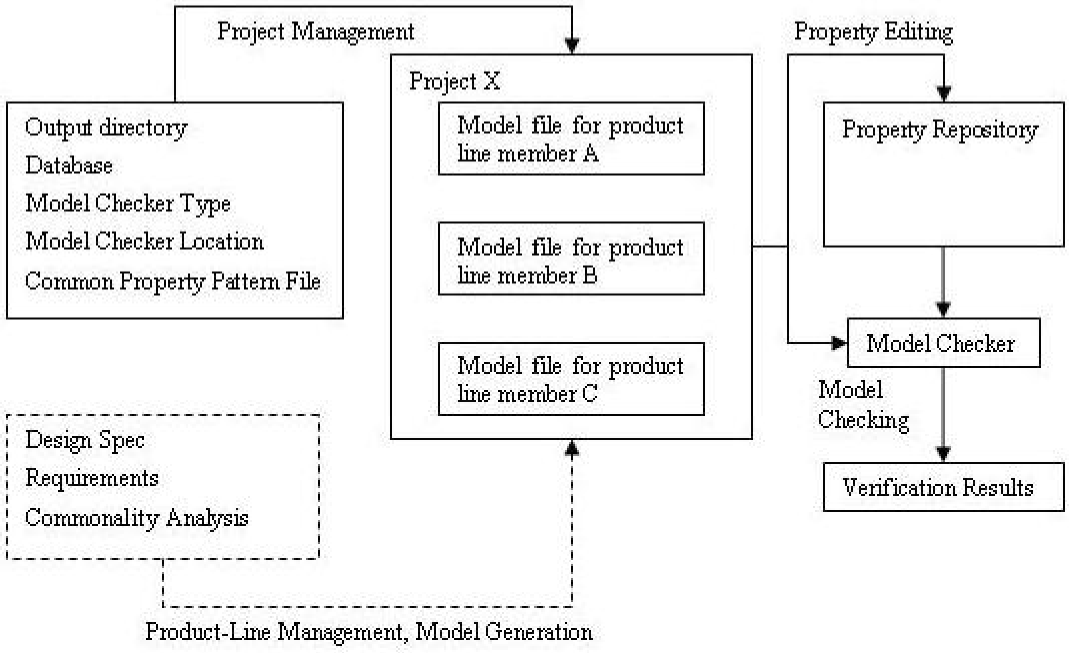
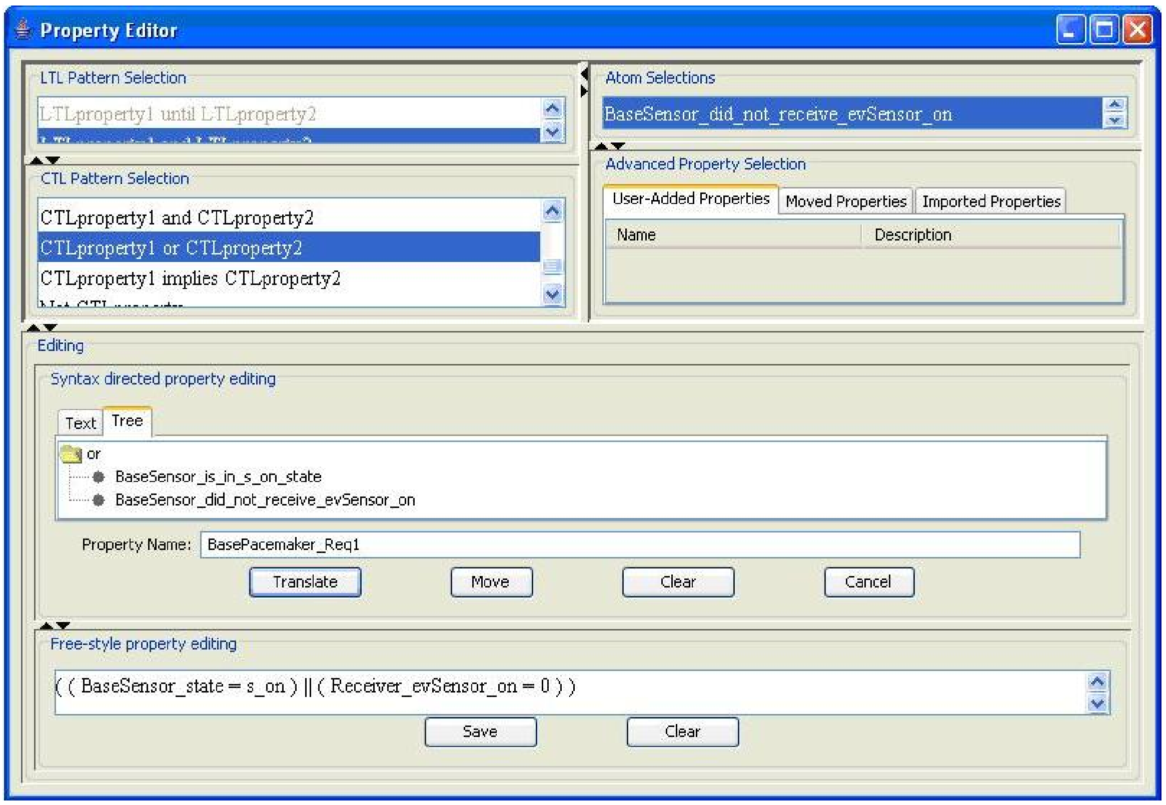


Figure 4.2 FormulaEditor architecture overview

system or product line, leaving out the ones that are no longer valid for the new system. This is accomplished by automatically detecting atoms in the property that do not belong to the system. Version control is enforced at single-system granularity by detecting and clearing out-of-sync verification results (i.e., a verification done before its model file changed is out of sync).

Property editing serves to generate product-line specific patterns and product-specific properties. Figure 4.3 shows the interface for property editing. It is divided into three areas: the upper area for selecting the building blocks of a property, the middle area for composing a property in natural language, and the lower area for composing or viewing a property as a temporal logic formula. The preset patterns are shown in the pattern selection part of the upper area. The default patterns are a complete set of basic LTL and CTL patterns that can be used to form any other LTL and CTL formulas [Huth and Ryan (2004)].

To provide traceability between the properties and the requirements, the tool also supports domain-customized natural language descriptions for properties and allows users to tag



LTLproperty1 until LTLproperty2

CTLproperty1 and CTLproperty2
CTLproperty1 or CTLproperty2
CTLproperty1 implies CTLproperty2

BaseSensor_did_not_receive_evSensor_on

or
BaseSensor_is_in_s_on_state
BaseSensor_did_not_receive_evSensor_on

Property Name: BasePacemaker_Req1

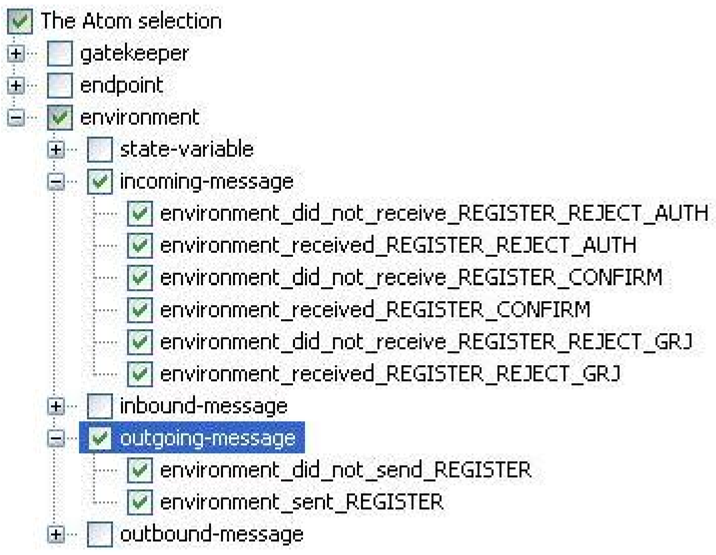
((BaseSensor_state = s_on) || (Receiver_evSensor_on = 0))

Figure 4.3   Property editing overview

properties with the requirements from which they are derived.

In the product-line domain-engineering phase (i.e., generating product-line reusable assets [Weiss and Lai (1999)]), users can create parameterized properties from the set of preset patterns (e.g., basic LTL and CTL patterns or Property Specification Patterns [Dwyer et al. (1999)] that recur in the product line). Those new parameterized properties then can be added to the preset patterns (stored in a common-pattern-file). The patterns can be reused for property specification both within the same system and across different systems in the product line.

In the product-line application-engineering phase (i.e., generating product-specific assets [Weiss and Lai (1999)]), users can instantiate the above-mentioned patterns (both preset ones and the ones created by the user on-the-fly) by replacing the parameters in the pattern with



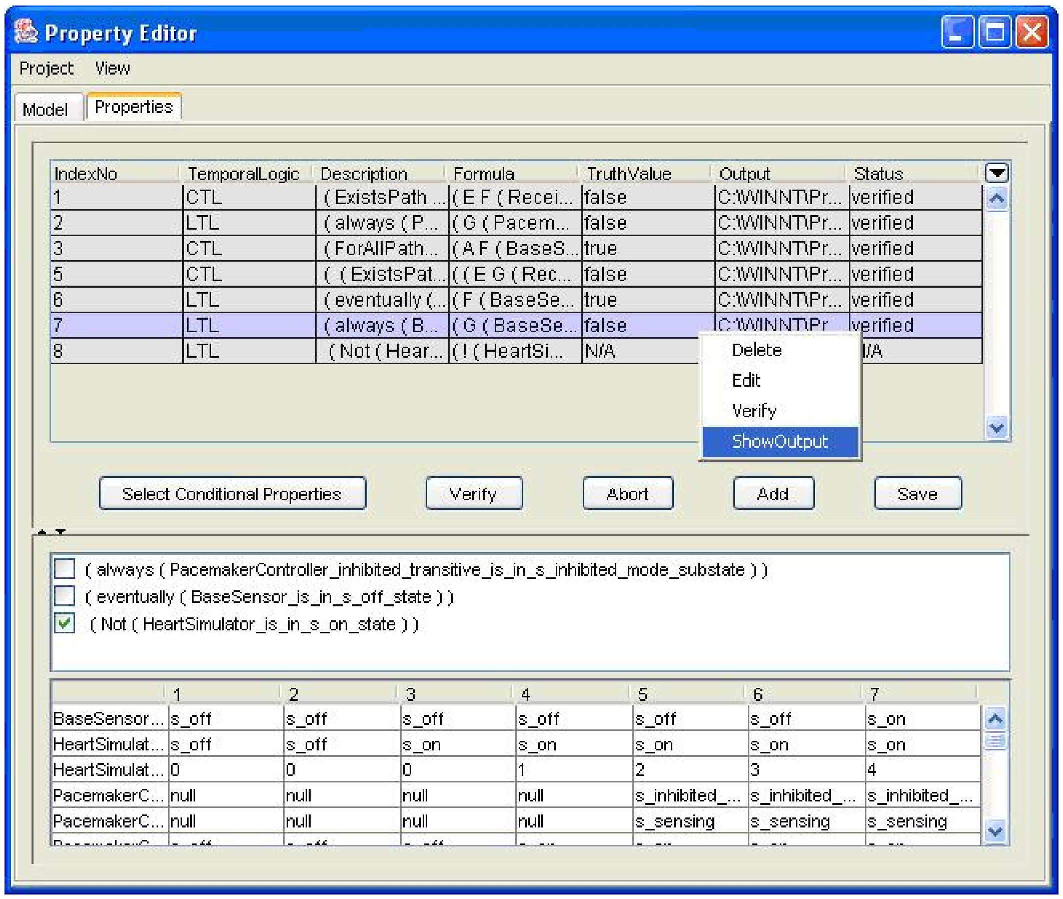
- [✓] The Atom selection
  - [ ] gatekeeper
  - [ ] endpoint
  - [✓] environment
    - [ ] state-variable
    - [✓] incoming-message
      - [✓] environment_did_not_receive_REGISTER_REJECT_AUTH
      - [✓] environment_received_REGISTER_REJECT_AUTH
      - [✓] environment_did_not_receive_REGISTER_CONFIRM
      - [✓] environment_received_REGISTER_CONFIRM
      - [✓] environment_did_not_receive_REGISTER_REJECT_GRJ
      - [✓] environment_received_REGISTER_REJECT_GRJ
    - [ ] inbound-message
    - [✓] outgoing-message
      - [✓] environment_did_not_send_REGISTER
      - [✓] environment_sent_REGISTER
    - [ ] outbound-message

Figure 4.4 Atom selection overview

concrete properties for the model file. They can also replace the parameters with other patterns to customize a generic pattern.

The tool provides two mechanisms to ensure that the instantiation process is done correctly. The first is to present users with a selection of properties to instantiate the patterns. These include the atomic properties (which we call "atoms", e.g., a variable defined in the model holding a value) extracted from the model as well as past specified properties and patterns for this model. The second mechanism is to provide a syntax-directed editing environment to prevent ill-formed properties from being created (e.g., mixing LTL with CTL properties) or saved (e.g., not-fully-instantiated properties, bad syntax) into the property pool for a specific product member.

For example, "(ForAllPaths always ((environment_sent_REGISTER) implies (ForAllPaths eventually CTLproperty)))" is the natural language description of a parameterized property derived from the Response Property Pattern [9], yet partially instantiated with the "environment_sent_REGISTER" message. The "LTLproperty" and the "CTLproperty" are parameters to be filled with specific properties in the application engineering phase. The "en-



IndexNo	TemporalLogic	Description	Formula	TruthValue	Output	Status
1	CTL	(ExistsPath ...	(E F (Recei...	false	C:\WINNT\Pr...	verified
2	LTL	(always (P...	(G (Pacem...	false	C:\WINNT\Pr...	verified
3	CTL	(ForAllPath...	(A F (BaseS...	true	C:\WINNT\Pr...	verified
5	CTL	((ExistsPat...	((E G (Rec...	false	C:\WINNT\Pr...	verified
6	LTL	(eventually (...	(F (BaseSe...	true	C:\WINNT\Pr...	verified
7	LTL	(always (B...	(G (BaseSe...	false	C:\WINNT\Pr...	verified
8	LTL	(Not (Hear...	(! (HeartSi...	N/A		I/A

- [ ] (always (PacemakerController_inhibited_transitive_is_in_s_inhibited_mode_substate))
- [ ] (eventually (BaseSensor_is_in_s_off_state))
- [x] (Not (HeartSimulator_is_in_s_on_state))

	1	2	3	4	5	6	7
BaseSensor...	s_off	s_off	s_off	s_off	s_off	s_off	s_on
HeartSimulat...	s_off	s_off	s_on	s_on	s_on	s_on	s_on
HeartSimulat...	0	0	0	1	2	3	4
PacemakerC...	null	null	null	null	s_inhibited_...	s_inhibited_...	s_inhibited_...
PacemakerC...	null	null	null	null	s_sensing	s_sensing	s_sensing

Figure 4.5 Model checking management overview

vironment_sent_REGISTER" message is selected from the "atom selection overview" window, popped out by clicking the "Atom Selections" in the upper area of Figure 4.3. A screenshot of the atom selection overview is shown in Figure 4.4.

FormulaEditor recognizes variable declarations in SMV models as atoms in a model. The tool then translates the atoms into its corresponding basic event description, following a set of translation rules approved by the domain engineers. In this way the atoms presented to a user are domain-customized descriptions, rather than their original forms in the model. For example, in the SELEX project, the variable Authenticate_In denotes an inbound micro-message, and we are only interested in whether the micro-message is consumed or not (meaning that Authenticate_In is zero or not). Therefore even though in the model Authenticate_In can range from 0 to 9, only two atoms are extracted, i.e., "Authenticate_In_consumed" and "Authenticate_In_waiting".

Model checking. For the model checking part of the tool, both the property and the model are fed into a model checker. (In the work described here, we used Cadence-SMV [McMillan (1993)] as the backend model checker, but we also have used CMU-SMV [Model Checking Group at CMU (2006)].) The tool provides the following facilities to help manage the model checking process in a product-line setting (as seen in Figure 4.5).

1. Automatically formats a generic LTL or CTL property to the form recognized by the back-end model checker; allows verification of multiple properties at a time.
2. Provides easy access to all information associated with a property (e.g., verification results, temporal logic type, counterexample, requirement it comes from etc.) and can group the properties according to different criteria (e.g., group by verification results, or group by requirements).
3. Supports conditional verification when it is allowed by the underlying model checker (Cadence-SMV in this case), i.e., users can select some existing properties as assumptions for another property. Those assumptions can be turned on or off before checking, and the enabled assumptions are an integral part of the verification result.

4.3 Case Study

In the previous sections we introduced our tool-supported technique based on the following two assumptions:

1) Property specification for model checking in an ad-hoc manner (without automatic linkage from the requirements to the models) is not likely to be reused as the system evolves.

2) Property specification for product lines without support for reuse can be tedious and hard to manage.

The claim is that since our technique overcomes the above two obstacles, it allows product-line property specification being conducted in a more efficient and sustainable manner.

In this section we step through one case study that we used to test the above claim. Since the focus of the technique is on property specification and management, we do not address model generation here.

4.3.1 Pacemaker Product Line

The second application of FormulaEditor was to a cardiac pacemaker product line, as introduced in Section 2.2.

In previous work [Liu et al. (2005a, 2007a,b)], we developed state machine models for each of the products in this product line using UML statecharts. We chose Cadence-SMV as our model checker because of the existing technique for translating UML statecharts to Cadence-SMV (e.g., [Van Langenhove and Hoogewijs (2004)]). The model translation is currently done manually. We take the safety properties to check from the systems' safety requirements.

The reason for choosing this product line as our case study is that it is an example of a safety-critical product line that would benefit from the added assurance of formal verification. In addition, this product line keeps growing, so being able to maintain traceability among properties as the models evolve and manage the re-verification process are essential if the project is to make a smooth transition into model checking from model-based development.

We model the pacemaker as a component-based system [Atkinson et al. (2002)] , meaning that components serve as the functional units that are composed for each product. The following components were modeled in Cadence-SMV: a sensing component (BaseSensor) that senses heart beat, a stimulation component (PulseGenerator) that generates pulses to the heart, a controlling component (PacemakerController) that configures different pacing and sensing algorithms and issues commands, MotionSimulator and HeartSimulator components that simulate the patient's motion and heart beat activities respectively, and a ExtraSensor component that senses a patient's activity level.

4.3.2 Product Line Property Specification

In this section, we describe the entire model-checking process, but focus on Steps 3-5 which use the FormulaEditor tool.

1. *Obtain the set of safety critical requirements that need to be model checked.*

Sample Requirements for BasePacemaker:

R1. When PacemakerController is in Sensing state and it receives a sensed event from BaseSensor, then PacemakerController shall command the BaseSensor to turn off at the next time unit.

R2. Whenever PacemakerController is in Refractory state, PulseGenerator shall remain in idle state.

R3. When PacemakerController is in Sensing state and the SenseTime is up, the PacemakerController shall command the BaseSensor to turn off at the next time unit.

Sample Requirements for ModeTransitivePacemaker:

R1, **R2**: same as BasePacemaker

R3.1 In Inhibited mode, when PacemakerController is in Sensing state and the SenseTime is up, the PacemakerController shall command the BaseSensor to turn off at the next time unit.

R3.2 In Triggered mode, when PacemakerController is in Sensing state and there is no heartbeat sensed by BaseSensor, the PacemakerController remains in Sensing state the next time unit.

2. *Conduct a commonality analysis [Weiss and Lai (1999)] of the requirements to find recurring requirements.*

We show such requirements here with the same number in different products (e.g., R1 in BasePacemaker and R1 in ModeTransitivePacemaker) as common requirements and requirements with associated numbering (e.g., R3.1 and R3.2 in ModeTransitivePacemaker) as a variation.

3. *Starting from the product that has the fewest variabilities, build a core set of properties and patterns to be reused from its requirements.*

We started from BasePacemaker, as all its requirements appeared in whole (i.e., R1, R2, R5) or in part (R3, R4) in the other three products. By adopting an incremental process, specifying the product with fewer variabilities first can maximize the reusability of any specified property set. Each time we add a new property, we first scan through the list of properties already specified or imported to see if we can reuse any existing ones by minimal modification effort. If not, we scan through the patterns to find one that we can instantiate with minimal instantiation effort. If we compose a new pattern out of existing patterns for a new property, we save that pattern to the common pattern file for product line reuse.

For example, the following pattern set was built for BasePacemaker (we use the natural language description here rather than the temporal logic formula because that is the format we view when we search for possibility of reuse):

Pattern 1:(always ((LTLproperty1 and LTLproperty2) implies (next LTLproperty)))

Pattern 2: (always (LTLproperty1 implies LTLproperty2))

Pattern 1 was built while specifying the property for R1. Pattern 2 was built while specifying the property for R2. These patterns were subsequently reused in the property specification for other requirements. When we specified properties for R3 and R5, we used Pattern 1 and Pattern 2 separately. When we specified property for R4, we reused property for R3 with one replacement of an atom in it.

4. *For each product, build its own property set by importing likely property sets previously specified. These are then reused and the product's own variations added (i.e., modifying properties and instantiating patterns).*

For example, both ModeTransitivePacemaker and RateResponsivePacemaker share requirements R1, R2, and R5 with BasePacemaker. They differ only in R3 and R4 by introducing their own variations: Inhibited and Triggered mode for ModeTransitivePacemaker, LRLrate and URLrate for RateResponsivePacemaker.

Thus, for each product, we imported the entire property table of BasePacemaker and then modified the properties for R3 and R4 respectively. By doing this, we shared the property set built for BasePacemaker. We also shared the pattern set because the patterns identified while specifying properties for BasePacemaker were saved to the common pattern file and became part of the property editing interface.

For ModeTransitive-RateResponsivePacemaker, we first imported the property tables for ModeTransitivePacemaker and then appended the property table for RateResponsivePacemaker, deleting any repeated properties and renaming different properties that happened to have the same name (the deleting and renaming process was manually done). In contrast with importing the property table only from the BasePacemaker, or only from ModeTransitivePacemaker or RateResponsivePacemaker, being able to import multiple tables had the benefit of maximizing the property pool for ModeTransitive-RateResponsivePacemaker. A discussion of this is given in Section 4.4.

5. *Invoke the underlying model checker to check all properties in the property set for each product. This was done in both incremental mode as the property set was being built and in batch mode to verify or re-verify as commonalities were imported.*

Step 5 interleaves with step 4 in that for each property set built, we model check it with Cadence-SMV right away. This is very important since we build the property set as the variabilities accumulate, so that a flaw detected in the model or in the property for one product will not propagate to other products that share its requirements or designs.

4.4 Discussion of Results

In this section, we discuss the effort saved, the different types of reuse, and the utilization of verification results while using this technique, as well as the support for evolution, and mapping from requirements to properties provided by this technique.

1. *Effort saved by using this technique*

At the beginning of the process (step 1 in the above section), we identified 28 requirements to be model-checked. In BasePacemaker, by the end of the process, 2 property patterns were identified, 2 properties were specified by instantiating those two patterns, and 1 property was specified by reusing another property specified for BasePacemaker with some modifications. In the other three products, 3 properties were specified by reusing existing properties in BasePacemaker without any modification. In ModeTransitivePacemaker and RateResponsivePacemaker, an additional 3 properties were specified by reusing BasePacemaker properties with some modifications. In ModeTransitive-RateResponsivePacemaker, 6 properties were specified by reusing ModeTransitivePacemaker properties with some modifications.

Modifying previously specified properties is similar to changing the instantiation of patterns as both the old and the new properties share similar structures. Therefore, among the property specification for the 28 requirements, 17 of them (about 61%) actually benefited from the product-line specific pattern approach.

2. *Pattern reuse and property reuse*

There are two types of reuse that were useful in this case study: pattern reuse and property reuse. Both can be used for specifying properties for a single product or across a product line. However, property reuse has the limitation that a property to be reused cannot contain any variables not defined in the underlying model. Thus, property reuse is more suitable for commonalities within a product-line scope, and carries the assumption that models for different products in the same product line adhere to a consistent naming conventions. Pattern reuse, on the other hand, does not have the above limitation and can be reused in any system. However, pattern reuse usually requires more instantiation effort than property reuse because it is more general.

A possible solution is to have partially instantiated patterns for reuse, i.e., we can replace the variation part in a property by generic parameters, or instantiate the common part in a pattern by atoms shared by all products.

3. *Utilization of verification results*

Two types of verification results were found helpful in identifying flaws:

1) Properties derived from requirements that were supposed to be true but produced counterexamples

In this situation, the counterexamples provided by the model checker were helpful in locating the faults. For example, the property for R2 in the ModeTransitive-RateResponsivePacemaker was initially shown to be false. The counterexample showed that the PulseGenerator stayed in Pulsing state after generating a pulse, indicating an error in the state-transition logic for the PulseGenerator component. An inspection of the model revealed an inconsistency when translating from UML-statecharts to Cadence-SMV code.

The counterexample itself sometimes revealed the problem without going back to the model. For example, when we changed "PacemakerController shall command the BaseSensor to turn off at the next time unit" in R1 for BasePacemaker to "BaseSensor shall be in the off state at the next time unit", the property derived from this modified requirement was shown to be false. The counterexample showed that the state-change for BaseSensor lagged two time units behind the command-issuing in PacemakerController. Thus, either the requirement had to be modified to be "When PacemakerController is in Sensing state and it receives a sensed event from BaseSensor, then PacemakerController shall command the BaseSensor to turn off at the next three time unit" in order to be valid for the system, or the system design (the model here) had to be changed.

The counterexamples showed the values that each involved variable took in a trace leading to the false result. Currently, variable names shown in counterexamples are the same as in the model. Thus, good naming conventions helped locate faults in the system design. We realize that there currently exists a gap between variable names in the counterexample and variable declaration in the atom-selection, as the latter is sugar-coated by giving domain-customized descriptions. We plan to address this issue by interpreting counterexample in terms of atoms.

2) Properties derived from variations of requirements that were supposed to be false but verified true

Verifying variations of requirements helped identify several vacuously true properties [Armoni et al. (2003)], i.e., properties that hold due to undesired reasons. For example, if we were to change requirement R4.1 for RateResponsivePacemaker from "when PacemakerController is in LRL (normal rate) state and in Sensing state, and the SenseTime is up, the PacemakerController shall command the PulseGenerator to generate a pulse at the next time unit" to "when PacemakerController is in LRL (normal rate) state and in Sensing state, and the SenseTimeShort is up, the PacemakerController shall command the PulseGenerator to generate a pulse at the next time unit" and it still held for the system, then we would need to check if the PacemakerController always commanded the PulseGenerator to generate a pulse regardless of SenseTime up or not.

In this case study, we changed different atom instantiations when reusing a pattern or property, to detect potentially vacuously true properties. As we mentioned before, the benefit of maximizing the property pool was for exactly this reason, since two similar properties are not necessarily both true in the same model.

An added value of the tool-supported technique was in detecting hidden requirements by testing different variations of one requirement. The tool makes it convenient to instantiate different property variations. Moreover, those variations specified for one requirement in a product could be shared via property-table-import by other products if they had a similar requirement, and be quickly verified using batch-verification.

4. *Support for evolution*

1) Product-line change adaptation. Product lines evolve quickly, so to be effective, a tool must readily accommodate frequent changes. We categorize possible changes to a product line and discuss how FormulaEditor handles each of them: (a) New commonality: augment the core property/pattern set to include the new commonality, and in each product in the product line, import the property/pattern derived from this commonality and make modifications when necessary. (b) New variability: augment the property set for all affected products to include this new variability, and augment the common pattern file if new patterns are introduced during property-derivation for such a variability. (c) Delete/modify existing commonality: update the core property set, and update the property set for every product in the product line. The common pattern file may be updated as well if new patterns are introduced during modification. (d) Delete/modify existing variability: update the property set for all affected products. The common pattern file may be updated as well if new patterns are introduced.

2) Model change adaptation using conditional verification. Note that sometimes the property update is done ahead of the model update, i.e., the model has not yet incorporated the new requirement, since the update of the model usually takes more time than updating the

properties. The updated added properties can then be set as conditional properties that are assumed to be true for the incomplete model. Existing properties can continue be verified given those conditional properties. Once the model is completed, we restore those properties as normal properties and verify them as before.

3) Property change adaptation. One restriction of the approach mentioned in the chapter is that while it is useful in incrementally building a property set for each product in a product line, changes in properties require manually updating each property set affected. Importing property tables can speed this process, but can still be labor-intensive given a large product line context.

A better solution could be to specify just the composition of a property specification from parameterized building blocks and automatically instantiate the full property specification once we instantiate the decision model for the product line [Weiss and Lai (1999)]. Geppert, Li, Rößler, and Weiss [Geppert et al. (2004)] have successfully applied the idea to test-case specification generation for product lines.

5. *Mapping from requirements to properties*

One benefit of property-editing tools (see the Related Work section) is to help users develop formal, unambiguous requirements suitable for model checking. This, when reflected in our technique, is detecting one-to-many mappings between requirements and atoms. For example, the initial requirement R1 for BasePacemaker was “when PacemakerController is in Sensing state and it receives a sensed event from BaseSensor, then BaseSensor should be turned off immediately”. The requirement ”BaseSensor should be turned off” is associated with a chain of events in the model, i.e., “PacemakerController sent evSensor off”, “BaseSensor received evSensor Off” and “BaseSensor is in off state”. Those events, although all mapping to the same requirement, do not happen within the same step. Thus, instantiating the same pattern with different atoms generated different verification results. Tying the property specification to the model helped us identify potential ambiguities (since the atoms for instantiating the patterns are all extracted from the model). With the backend model checker integrated into the tool, we were able to quickly verify a group of properties to get detailed information about how to dismiss the ambiguity, e.g., by changing “BaseSensor should be turned off immediately” to “PacemakerController shall command the BaseSensor to turn off at the next time unit”.

The insight gained by inspecting one-to-many mappings can also help generate accurate properties for more complex component-based systems or telecommunication protocols, where the delay in response-chain inside components or in network communication can significantly affect the validity of certain requirements.

Another benefit is that by reusing properties and patterns in the product-line scope, users can significantly reduce the effort of deriving the right properties from requirements. In addition, if a pattern is found to be incorrect, then the properties instantiated from it need to be corrected accordingly.

6. *Possible applications*

Although this technique starts from a product-line point of view, it is not confined to use in a product line setting. We have, e.g., begun applying our technique to NASA’s Simplified Aid for EVA Rescue (SAFER) system [Vito (2000)]. Initial results suggest that this technique can help re-engineer existing property specifications for legacy systems to support reuse in future system developments.

CHAPTER 5. COMPOSITIONAL MODEL CHECKING

This chapter introduces a new technique for incremental and compositional model checking that allows efficient reuse of model-checking results associated with the features in a product line. As the use of product lines has increased, so has the need to verify the models used to construct the products in the product line. However, this effort is currently hampered by the difficulty of composing model-checking results for the features in a way that allows reuse for subsequent products. The contributions of this work are a formal framework for the model-checking technique that removes restrictions on how the features can be sequentially composed, and a description of how to generate obligations at variation points such that composed features can be more easily verified as new products are built. This work[1] develops the technique and its implementation in the context of a medical-device product line.

5.1 Introduction

Software product lines are widely used due to their advantageous reuse of shared elements, but this reuse across different products poses challenges for model checking of product lines. Especially for high-integrity product lines such as pacemakers, medical imaging systems, and avionics control systems, we would like to use model checking to verify that key properties hold in each new product. However, model-based verification of software product lines is currently hampered by the complexity of composing model checking results of the various features in a way that allows reuse when model checking new products.

In a software product line, the products all share a common set of mandatory features

[1]The work presented in this chapter is adapted from Liu, J., Basu, S., and Lutz, R. (2008a). Generating variation-point obligations for compositional model checking of software product lines. Technical Report, 08-04, Computer Science, Iowa State University. It has been submitted for journal review.

but are differentiated one from the other by their variable (optional and alternative) features [Weiss and Lai (1999)]. Each feature carries an increment of functionality for the system [Batory et al. (2006)]. Typically, a set of variations are selected and composed on top of the common base features to create each distinct, new product [Jacobson et al. (1997); Webber and Gomaa (2004)]. The locations in the features where other features can be added to construct the various products are called *variation points*.

Model checking [Clarke et al. (2000); Huth and Ryan (2004)] takes a model of a given system's design, and checks if it satisfies certain properties of the system, interpreted in terms of logic formulas. It is a powerful technique for enhancing the quality of software systems, e.g., by identifying flaws that would not have been caught otherwise [Havelund et al. (2001); Kaivola (2005)]. As such, model checking can play a vital role in verifying key properties of products in high-integrity product lines such as pacemakers, medical imaging systems, and avionics control systems.

However, formal reasoning about each product in isolation fails to exploit the fact that all the products in a product line share common features. Similarly, many products in a product line will share some of the variable features. Repeated verification of the same sets of features wastes resources and discourages industrial adoption of model checking for product lines.

This work presents an incremental and compositional model checking technique that allows reuse of model checking results associated with the features in a product line. The contribution of this work is that, in contrast with existing work on compositional model checking of features [Blundell et al. (2004); Wang (2005); Thang (2005)], we impose no restrictions (e.g., regarding the sequence, type of connection points, or number of connections) on how the features can be composed. Any type of sequential composition of features, not just pipelined composition, can be verified. Similarly, behavior of a feature that depends on another feature's behavior (which may, in turn, depend on the first feature) can be verified. Extending previous work [Liu et al. (2008a)], the resulting composition is shown to be sound and complete.

Our compositional model checking technique generates obligations at the variation points such that the feature composition satisfies the desired property if and only if the features added at variation points satisfy the corresponding obligations. These *variation point obligations* guide the verification of features subsequently composed at the variation points as new products in the product line are built. The algorithms that generate the obligations incorporate optimization strategies to reduce the unnecessary model checking load.

By allowing more kinds of interactions between features, our approach provides three important advantages for model checking product lines. First, such flexibility means that many more real-world systems can be model-checked. This moves model checking closer to product-line development practice. Second, the implementation stores the variation point obligations obtained for each feature during earlier model checking runs, thus enabling reuse of previous model checking results when a new product is composed. Re-verification is only performed when needed, providing savings in space and time over non-compositional model checking. Third, as a product line evolves, new variation points are typically introduced. The technique described in this chapter accommodates such changes by identifying obligations at these new variation points from previous obligation computations done at those points of change. We have implemented the technique, and demonstrate and evaluate it on a medical device product line.

The rest of the chapter is organized as follows. Section 5.2 presents some background and related work, and Section 5.3 provides a motivating example. Section 5.4 describes preliminary information. Section 5.5 illustrates each step of the compositional model checking. Section 5.6 presents the correctness proof of the algorithms used in this technique, and Section 5.7 provides some useful implementation details. Section 5.8 demonstrates our technique on a simplified pacemaker product line and discusses test results. Finally, Section 5.9 discusses the reuse effectiveness and applicability of this work.

5.2 Background

Feature-based modeling has been widely explored both for the development of software systems, e.g., [Zave (1993); Batory et al. (2006)] and for product lines, e.g., [Kang et al. (2002)]. Each feature carries an increment of functionality for the system [Batory et al. (2006)].

A major benefit is that features "help localize the effects of adding or making changes to units of functionality"[Ossher and Tarr (1999), p.4]. Feature modeling has been especially useful for product-line development because a feature-based model "provides a basis for developing, parameterizing, and configuring reusable assets"[Kang et al. (2002), p.58].

The concept of *feature* is differentiated from the concept of *component*. While both can model a functional unit or a service, a *feature* often denotes "an end-user visible characteristic of a system"[Pohl et al. (2005), p.92], while a *component* does not have to be end-user visible, but is a modular unit with well-defined interfaces and can be deployed separately [Szyperski (1998)]. Thus, feature is closely tied to the requirements and reflects more of the user's point of view while component is closely tied to the implementation, e.g., in the form of objects or collections of objects, and reflects more of the engineer's point of view. Though we use both notions in the running example, we use feature models to check the product line properties as it is more natural to align the verification with the product-line requirements.

Several recent works have investigated representations of variability within a product line in behavioral models. Fantechi and Gnesi identify whether a product belongs to a product line [Fantechi and Gnesi (2007)]. Fischbein, Uchitel and Braberman propose a technique to determine whether a variability undermines a product-line property [Fischbein et al. (2006)]. Lauenroth and Pohl describe how variability complicates the consistency checking of a product in the product line [Lauenroth and Pohl (2007)]. This line of work does not treat the common and variable functionality as equal units to be composed. Instead, they have a well-defined base with relatively small variations (e.g., transitions in a finite state machine) that may be verified through techniques like behavioral conformance [Fischbein et al. (2006)]. The variations that they can verify do not cover all the ones that exist in our case.

5.3 Illustrative Example

The work reported here was motivated by the difficulty of reusing model-checking results during the development and evolution of safety-critical product lines. Our effort is directed at enabling reuse of previous model-checking results so that system properties can be efficiently verified when a new product is built in the product line. This work uses an example of a simplified pacemaker product line, introduced in Section 2.2, to evaluate performance of our approach (Sect. 5.8).

Certain properties must be shown to be true for every product in the pacemaker product line in order to assure patient safety. An example of such a property is: *In the InhibitedMode, the pacemaker shall always generate a pulse when no heartbeat is detected during the normal sensing interval.*

Since verification of this property involves BasePacemaker functionality common to all the products, we would like to avoid unnecessary, repeated checking of the same feature as each new product is built. We next describe our approach to achieving this through appropriate reuse of the results from previous model-checking runs. By generating and managing obligations at the variation points, the model-checking effort is aligned with the inherent variation points that a product-line development approach provides.

5.4 Preliminaries

In this section, we describe the preliminaries of this technique, including the formal modeling of the functional behavior of features, and the formal specification of the functional requirements.

5.4.1 Feature Behavioral Modeling

We represent the functional behavior of features in a product line using finite state machines where states represent the configurations of the functional behavior, and transitions from one state to another represent the evolution of the behavior between configurations. Formally, the model is defined as follows:

Definition 1 (Feature Behavioral Model)**.** *Feature behavioral model* $\mathtt{FM} = (S, S_0, V, T, L)$ *where* S *is the set of states,* $S_0 \subseteq S$ *is the set of initial states,* $V \subseteq S$ *is the set of variation points,* $T \subseteq S \times S$ *is the transition relation, and* $L : S \rightarrow 2^P$ *is the labeling function which* *associates each state $s \in S$ with the set of propositions in P that are true in that state. We will denote $(s, s') \in T$ by $s \rightarrow s'$.* □

In the above, $s \in V$ acts as the variation point where one `FM` can be plugged into another, i.e., when two `FM`s are *sequentially* composed, new transitions are added from some variation points of one to states in the other. For each composition between $\mathtt{FM}_1$ and $\mathtt{FM}_2$, we use $T_c^{\mathtt{FM}_1,\mathtt{FM}_2} \subseteq V^1 \times S^2$ where V^1 is the set of variation point of $\mathtt{FM}_1$ and S^2 is the set of states in $\mathtt{FM}_2$. The relation $T_c^{\mathtt{FM}_1,\mathtt{FM}_2}$ denotes how the states in $\mathtt{FM}_2$ are connected to the variation points of $\mathtt{FM}_1$. We define sequential composition as follows:

Definition 2 (Sequential Composition). *Given $FM_1 = (S^1, S_0^1, V^1, T^1, L^1)$, $FM_2 = (S^2, S_0^2, V^2, T^2, L^2)$, $T_c^{FM_1,FM_2}$, and $T_c^{FM_2,FM_1}$, the sequential composition $Comp_{seq}(FM_1, FM_2) = (S^1 \cup S^2, S_0^1, V^{12}, T^{12}, L^{12})$,*

1. $V^{12} = V^1 \cup V^2$,

2. $T^{12} = T^1 \cup T^2 \cup T_c^{FM_1,FM_2} \cup T_c^{FM_2,FM_1}$, *and*

3. $L^{12}(s) = \begin{cases} L^1(s) & \mathtt{if}\ s \in S^1 \\ L^2(s) & \mathtt{otherwise} \end{cases}$ □

Observe that the above definition allows $\mathtt{FM}_1$ to be connected to $\mathtt{FM}_2$ and vice versa, resulting in possible loops between the behaviors of the two features. $\mathtt{FM}_1$ is the start feature at whose start states (S_0^1) we want a given property to be satisfied. The set of variation points V^{12} of $Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2)$ includes the states in V^i $(i \in \{1, 2\})$ that may have been used in the composition. They are also the states that can be used as variation points for future additions of other features.

A *closed* `FM` is one which does not have any variation points $(V = \emptyset)$. In other words, a closed `FM` cannot be augmented with new features. An open `FM` is one whose set of variation points is non-empty. An open `FM` is important for a feature to be reused in the product-line setting, as it can be composed with different features in different products.

5.4.2 Temporal Logic CTL

Properties are described using Computation Tree Logic (CTL) [Huth and Ryan (2004)]. We present a brief overview of the syntax and semantics of CTL formulas. The syntax of CTL can be defined as follows:

$$\phi \;\rightarrow\; \texttt{tt} \mid \texttt{ff} \mid P \mid \neg\phi \mid \phi \vee \phi \mid \texttt{EX}(\phi) \mid \texttt{E}(\phi\ \texttt{U}\ \phi) \mid \texttt{EG}(\phi)$$

The semantics of the CTL formulas are given in terms of the states of finite state systems (FM) that satisfy the formulas. The propositional constant tt is satisfied in all states while ff is not satisfied by any state. The proposition p $(\neg p)$ is satisfied by state s such that $p \in L(s)$ $(p \notin L(s))$. $\neg\varphi$ is satisfied by states where φ is not satisfied. $\varphi_1 \vee \varphi_2$ is satisfied by states that satisfy φ_1 or φ_2. $\texttt{EX}(\varphi)$ is satisfied by a state which has at least one transition to a state that satisfies φ. $\texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2)$ is satisfied by a state which has a path where φ_1 holds in every state in that path until a state satisfying φ_2 is reached. $\texttt{EG}(\varphi)$ is satisfied by a state which has a path where every state in the path satisfies φ.

The above syntax forms the adequate set of CTL formula syntax. Some other widely used syntactic constructs such as $\texttt{EF}(\varphi), \texttt{AX}(\varphi), \texttt{AF}(\varphi), \texttt{A}(\varphi_1\ \texttt{U}\ \varphi_2), \texttt{AG}(\varphi)$ can be obtained from the adequate set; for example: $\texttt{AX}(\varphi) \equiv \neg\texttt{EX}(\neg\varphi)$ and $\texttt{EF}(\varphi) \equiv \texttt{E}(\texttt{tt}\ \texttt{U}\ \varphi)$.

A state belonging to the semantics of φ implies that the state satisfies φ, denoted by $s \models \varphi$. We say that a closed $\texttt{FM} = (S, S_0, \emptyset, T, L)$ satisfies a CTL formula φ, denoted by $\texttt{FM} \models \varphi$, if and only if $\forall s \in S_0 : s \models \varphi$. For a detailed discussion of CTL model checking closed system see [Huth and Ryan (2004)].

5.5 Detailed Approach

In this section we first provide an overview of the compositional model checking technique, followed by a detailed description of each of its steps. As noted in Section 5.4, a closed FM can be verified immediately to check whether or not it satisfies a desired CTL property. However, for an open FM such as ours, satisfiability of CTL properties may depend on the behavior of the features being connected to its variation points.

Given an $\texttt{FM}$ and a desired property for its possible compositions with other $\texttt{FM}$s, our solution relies on generating a set of CTL formulas as obligations for each of its variation points. A composition satisfies the desired property if and only if the added features at each variation point satisfy the corresponding obligations. We refer to these obligations as *variation point obligations*. As our definition of the feature composition allows loops between the features, such circular dependency is handled by recording in a global database, *answer set* (denoted by $\texttt{aSet}$), whether or not variation point obligations are satisfied by a composition.

Thus, checking whether a sequential composition $Comp_{seq}(\texttt{FM}_1, \texttt{FM}_2, \cdots, \texttt{FM}_m)$ satisfies a CTL formula φ amounts to checking whether all the start states of $\texttt{FM}_1$ satisfy φ and can be compositionally resolved as follows:

Step 1 Generate the variation point obligations for satisfying φ in all the variation points of $\texttt{FM}_i$ (initially $i = 1$). Record the variation point obligations in $\texttt{aSet}$.

Step 2 Use $T_c^{\texttt{FM}_i,\texttt{FM}_k}$ to identify all the features $\texttt{FM}_k$s connected to the variation points of $\texttt{FM}_i$: if state t_k in $\texttt{FM}_k$ is connected to variation point s_i of $\texttt{FM}_i$, where the variation point obligation is φ_i, iterate from Step 1 (with $i = k$) to compute the variation point obligations for each $\texttt{FM}_k$, such that t_k satisfies φ_i. If t_k satisfies its obligation φ_i, then update the $\texttt{aSet}$ entry for s_i in $\texttt{FM}_i$.

Step 3 If the $\texttt{aSet}$ cannot be further updated from computing variation point obligations, break from the iteration. Analyze $\texttt{aSet}$ to identify loops between features and update $\texttt{aSet}$ accordingly.

If the final $\texttt{aSet}$ records that the start states of $\texttt{FM}_1$ satisfy φ, then the composition $Comp_{seq}(\texttt{FM}_1, \texttt{FM}_2, \cdots, \texttt{FM}_m)$ satisfies φ.

In the rest of this section, we describe the generation of variation point obligations, how to update the answer set to identify inter-feature loops, and our algorithm for compositional model checking.

$$
\begin{array}{lrcl}
\text{A.} & \texttt{t\_Obl}(\varphi, s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) & := & \begin{cases} \psi & \text{if } (\varphi, s) \mapsto \psi \ \in \ \texttt{aSet}_{in}; \\ & \text{where } \texttt{aSet}_{out} := \texttt{aSet}_{in} \\ \\ \psi & \text{otherwise} \\ & \text{where } \psi'' := \texttt{Obl}(\varphi, s, H, \texttt{aSet}_{in}, \texttt{aSet}_{temp}) \\ & \texttt{aSet}_{out} := \begin{cases} \texttt{aSet}_{temp}[(\varphi, s) \mapsto \psi'/ \\ \qquad (\varphi, s) \mapsto \psi'' \ op \ \psi'] \\ \quad \text{if } (\varphi, s) \mapsto \psi' \in \texttt{aSet}_{temp} \\ \quad \text{where } op := \begin{cases} \wedge \ \text{ if } \varphi \text{ is} \\ \text{a universal} \\ \vee \\ \text{otherwise} \end{cases} \\ \quad \psi = \psi'' \ op \ \psi' \\ \\ \texttt{aSet}_{temp} \cup \{(\varphi, s) \mapsto \psi''\} \\ \quad \text{otherwise} \\ \quad\ \text{where } \psi = \psi'' \end{cases} \end{cases} \\
1. & \texttt{Obl}(\texttt{tt}, s, H, \texttt{aSet}, \texttt{aSet}) & := & (\texttt{tt}, \epsilon, \bot) \\
2. & \texttt{Obl}(\texttt{ff}, s, H, \texttt{aSet}, \texttt{aSet}) & := & (\texttt{ff}, \epsilon, \bot) \\
3. & \texttt{Obl}(p, s, H, \texttt{aSet}, \texttt{aSet}) & := & \begin{cases} (\texttt{tt}, \epsilon, \bot) & \text{if } p \in L(s) \\ (\texttt{ff}, \epsilon, \bot) & \text{otherwise} \end{cases} \\
4. & \texttt{Obl}(\neg\varphi, s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) & := & \neg\texttt{t\_Obl}(\varphi, s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) \\
5. & \texttt{Obl}(\varphi_1 \vee \varphi_2, s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) & := & \texttt{t\_Obl}(\varphi_1, s, H, \texttt{aSet}_{in}, \texttt{aSet}_{temp}) \ \vee \\ & & & \texttt{t\_Obl}(\varphi_2, s, H, \texttt{aSet}_{temp}, \texttt{aSet}_{out}) \\
6. & \texttt{Obl}(\texttt{E}(\varphi_1 \ \texttt{U} \ \varphi_2), s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) & := & \begin{cases} (\texttt{ff}, \epsilon, \bot) \quad \text{if } (\texttt{E}(\varphi_1 \ \texttt{U} \ \varphi_2), s) \in H; \\ \quad \text{where } \texttt{aSet}_{out} := \texttt{aSet}_{in} \\ \\ \texttt{t\_Obl}(\varphi_2 \vee (\varphi_1 \wedge \texttt{EX}(\texttt{E}(\varphi_1 \ \texttt{U} \ \varphi_2))), s, \\ H \cup \{(\texttt{E}(\varphi_1 \ \texttt{U} \ \varphi_2), s)\}, \texttt{aSet}_{in}, \texttt{aSet}_{out}) \\ \quad \text{otherwise} \end{cases} \\
7. & \texttt{Obl}(\texttt{EG}(\varphi), s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) & := & \begin{cases} (\texttt{tt}, \epsilon, \bot) \quad \text{if } (\texttt{EG}(\varphi), s) \in H; \\ \quad \text{where } \texttt{aSet}_{out} := \texttt{aSet}_{in} \\ \\ \texttt{t\_Obl}(\varphi \wedge \texttt{EX}(\texttt{EG}(\varphi))), s, \\ H \cup \{(\texttt{EG}(\varphi), s)\}, \texttt{aSet}_{in}, \texttt{aSet}_{out}) \\ \quad \text{otherwise} \end{cases} \\
8. & \texttt{Obl}(\texttt{EX}(\varphi), s, H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) & := & \bigvee_{s \to s'} \texttt{t\_Obl}(\varphi, s', H, \texttt{aSet}_{in}, \texttt{aSet}_{out}) \ \vee \ \begin{cases} (\varphi, s, \vee) \\ \quad \text{if } s \in V \\ (\texttt{ff}, \epsilon, \bot) \\ \quad \text{otherwise} \end{cases}
\end{array}
$$

Figure 5.1 Variation point obligations

5.5.1 Variation Point Obligations

Variation point obligations are sets of formulas associated with the variation points that must be satisfied by the features connected to them. The obligations are also annotated with the boolean operators $\wedge$ and $\vee$ to handle cases where multiple features are connected to the same variation point. They are formally defined as follows:

Definition 3 (Variation point Obligation)**.** *Given an* FM $= (S, S_0, V, T, L)$*, an obligation at a variation point is a formula* Ψ *in the following grammar:* $\Psi \rightarrow (\phi, s, op) \mid \neg\Psi \mid \Psi \vee \Psi \mid \Psi \wedge \Psi$ *where* ϕ *is a CTL formula,* $s \in V \cup \{\epsilon\}$*,* $op \in \{\vee, \wedge, \perp\}$*.* $\square$

The variation point obligation $(\varphi, s, \vee)$ states that one of the features added at the variation point (state s) must satisfy φ; $(\varphi, s, \wedge)$ means that each new feature added at state s must satisfy φ. A variation point obligation of the form $(\varphi_1, s_1, \vee) \vee (\varphi_2, s_2, \vee)$ (resp. $(\varphi_1, s_1, \vee) \wedge (\varphi_2, s_2, \vee)$) states that $(\varphi_1, s_1, \vee)$ or (resp. and) $(\varphi_2, s_2, \vee)$ must be satisfied. Finally, $\neg(\varphi, s, \vee) \equiv (\neg\varphi, s, \wedge)$ is satisfied at the variation point if φ is not satisfied in any of the new features added at s.

We use $(\varphi, \epsilon, \perp)$ to indicate that φ is not an obligation at any variation point. We also use the following simplification rules: $(\texttt{tt}, \epsilon, \perp) \vee \psi \equiv (\texttt{tt}, \epsilon, \perp)$; $(\texttt{tt}, \epsilon, \perp) \wedge \psi \equiv \psi$; $\neg(\texttt{tt}, \epsilon, \perp) \equiv (\texttt{ff}, \epsilon, \perp)$; $(\texttt{ff}, \epsilon, \perp) \wedge \psi \equiv (\texttt{ff}, \epsilon, \perp)$; $(\texttt{ff}, \epsilon, \perp) \vee \psi \equiv \psi$; $\neg(\texttt{ff}, \epsilon, \perp) \equiv (\texttt{tt}, \epsilon, \perp)$; $\neg(\psi_1 \vee \psi_2) \equiv \neg\psi_1 \wedge \neg\psi_2$; $\neg(\psi_1 \wedge \psi_2) \equiv \neg\psi_1 \vee \neg\psi_2$. We use $\psi, \psi_1, \psi_2, \ldots$ to denote variation point obligation formulas while using $\varphi, \varphi_1, \varphi_2, \ldots$ to represent CTL formulas.

Generation of variation point obligation follows in similar fashion to local CTL model checking Huth and Ryan (2004) where, given a state and a CTL formula to be satisfied at that state, the algorithm proceeds by recursively exploring the reachable state space and by unfolding the CTL formula. We use the following equivalences of CTL formulas for formula unfolding: $\texttt{E}(\varphi_1 \ \texttt{U} \ \varphi_2) \equiv \varphi_2 \vee (\varphi_1 \wedge \texttt{EX}(\texttt{E}(\varphi_1 \ \texttt{U} \ \varphi_2))$; $\texttt{EG}(\varphi) \equiv \varphi \wedge \texttt{EX}(\texttt{EG}(\varphi))$.

5.5.2 Step 1: Computing variation point obligations

Given an `FM` and a CTL formula φ, we define for every state s in `FM` the functions `t_Obl` and `Obl`, which generate the obligations at the variation points of `FM` required for s to satisfy

φ. The functions take five parameters: φ, the CTL formula that is required to be satisfied at s; s, the current state of the FM; H, the history set recording the state-formula pairs that have been visited in the recursive execution of the functions (to handle loops in the FM); $\texttt{aSet}_{in}$ and $\texttt{aSet}_{out}$ (the answer sets before and after the invocation of the function).

The answer set contains elements of the form $(\varphi, s) \mapsto \psi$ where φ is a CTL formula and ψ is a variation point obligation. We say that satisfiability of φ at s depends on the satisfiability of ψ. Specifically,

1. $(\varphi, s) \mapsto (\varphi', s', op')$ denotes that for s to satisfy φ, all (at least one of the) features connected via the variation point s' must satisfy φ' when op' is equal to $\wedge$ (resp. $\vee$).
2. $(\varphi, s) \mapsto (pc, \epsilon, \bot)$ denotes that s satisfies (does not satisfy) φ when pc is a propositional constant equal to tt (resp. ff)
3. $(\varphi, s) \mapsto \psi_1 \wedge \psi_2$ denotes that s satisfies φ if both ψ_1 and ψ_2 are satisfied. Similarly, $(\varphi, s) \mapsto \psi_1 \vee \psi_2$ denotes that s satisfies φ if one of ψ_1 and ψ_2 is satisfied.

The aSet is necessary to handle loops across multiple features (see Step 3 in Section 5.5.4). It also allows us to reuse the previous results to remove redundant computations. The recursive definition of the functions t_Obl and Obl is presented in Fig. 5.1.

Rule A corresponds to t_Obl (top-level call) which states that a variation point obligation corresponding to state s and formula φ is equal to the result present in $\texttt{aSet}_{in}$ if t_Obl has been invoked on the same state-formula pair before. If the current invocation of t_Obl is the first-time call with the corresponding state-formula pair, then Obl is invoked, and its result ψ'' is used to update the answer set. Note that the call to Obl may update the $\texttt{aSet}_{in}$ to $\texttt{aSet}_{temp}$. If the latter already contains an entry of the form $(\varphi, s) \mapsto \psi'$, then the mapping for (φ, s) is updated to $\psi''\ op\ \psi' (= \psi)$ where op is decided on the basis of the formula being universal (e.g. AG, AU) or existential (e.g. EG, EU)[2].

The choice of op can be explained as follows. ψ and ψ' are the variation point obligations that need to be satisfied for φ to hold at s. If φ is an universal (resp. existential) formula, the

[2]We are considering adequate set containing the existential path temporal formulas EG and EU; op will be disjunction in this case.

obligation at the variation point will also require all (resp. at least one) features connected to that variation point to satisfy that obligation. Accordingly, the result is obtained via conjunction or disjunction operation(s).

Rules 1–8 correspond to the `Obl` function. Observe that `Obl` invokes `t_Obl` to appropriately use the `aSet`. The first three rules in Fig. 5.1 state that for propositional constants and propositions, there is no obligation at the variation points; satisfiability of these types of CTL formulas can be decided at the current state s. As these function rules do not update the answer set, $\texttt{aSet}_{in}$ and $\texttt{aSet}_{out}$ remain unchanged. In Rule 5, the answer set updates are chained from one `t_Obl` call to the other.

Rules 6 and 7 use the history set H to decide the satisfiability of `EU` and `EG` properties in the presence of a loop (in the same feature model). If the state-formula pair is present in the history set, this shows circular dependency in a path. Thus, for the least fixed point formula `EU`, the result is $(\texttt{ff}, \epsilon, \bot)$. For the greatest fixed point formula `EG`, the result is $(\texttt{tt}, \epsilon, \bot)$. On the other hand, if the state-formula pair is not present in the history set, the formula is expanded to its equivalent form, `t_Obl` is invoked, and the history set is updated.

Finally, Rule 8 deals with $\texttt{EX}(\varphi)$ formulas. The obligation is computed by expanding or moving to all possible next states of the current state s. There are two disjuncts in the result. The first disjunct shows that for each $s \rightarrow s'$, `t_Obl` is computed using s' and φ, and the results are OR-ed. This is because $\texttt{EX}(\varphi)$ is satisfied at s if there *exists* one next state that satisfies φ. The second disjunct states that if s is a variation point, then one of its future next states (there could be one or several), which is a state of a new feature connected to it, will have the obligation to satisfy φ.

Example. *Fig. 5.2 shows three features with the behavior of each represented by a state with a self-loop. Inter-feature transitions are shown as broken lines. All states in the example are variation points. Given the CTL property* $\varphi = \texttt{E}(\texttt{tt}\ \texttt{U}\ p)$ *to verify over the three-feature composition* $Comp_{seq}(F_1, F_2, F_3)$*, we first compute the variation point obligation for* F_1*, shown in Fig. 5.3. The downward arrows in Fig. 5.3 show the invocation of the* `t_Obl` *and* `Obl` *functions, while the upward arrows show the updates to* `aSet`*. The variation point obligations for* F_2 *and*

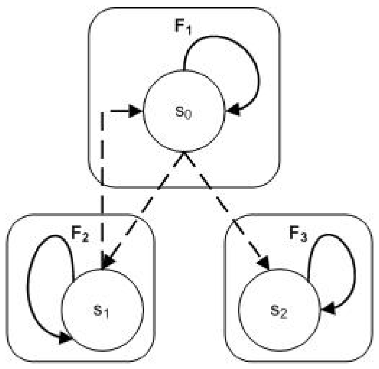


Figure 5.2 Feature composition example

F_3 are computed in a similar fashion.

5.5.3 Step 2: Updating the Answer Set

After the computation of variation point obligation terminates for one `FM`, `aSet` is updated with new results $(\texttt{tt}, \epsilon, \bot)$ and $(\texttt{ff}, \epsilon, \bot)$ in order to incorporate information regarding whether the state s satisfies a formula:

$$\begin{aligned}
&\texttt{update}(\texttt{aSet}) := \texttt{aSet}[(\varphi, s) \mapsto \psi/(\varphi, s) \mapsto \psi[\psi_i/\psi'_i]] \\
&\text{where } \psi_i := (\varphi_i, s_i, op_i) \text{ and } s_i \rightarrow t_i \ \in \ T_c^{\texttt{FM}_m, \texttt{FM}_n} \text{ and} \\
&\qquad (\varphi_i, t_i) \mapsto (pc, \epsilon, \bot) \in \texttt{aSet} \ \wedge \ pc \in \{\texttt{tt}, \texttt{ff}\} \text{ and}
\end{aligned}$$

$$\psi'_i = \begin{cases} \psi_i & \text{if } (pc = \texttt{tt}) \wedge (op_i = \wedge) \\ \psi_i & \text{if } (pc = \texttt{ff}) \wedge (op_i = \vee) \\ (\texttt{tt}, \epsilon, \bot) & \text{if } (pc = \texttt{tt}) \wedge (op_i = \vee) \\ (\texttt{ff}, \epsilon, \bot) & \text{otherwise} \end{cases}$$

The function states that the entry $(\varphi, s) \mapsto \psi$ in `aSet` is updated to $(\varphi, s) \mapsto \psi[\psi_i/\psi'_i]$ (ψ_i, a subformula of ψ, is replaced by ψ'_i). ψ_i is a variation point obligation of the form (φ_i, s_i, op_i) that was computed for a $\texttt{FM}_m$. If the state t_i of $\texttt{FM}_n$ is connected to the variation point s_i and there exists an entry $(\varphi_i, t_i) \mapsto (pc, \epsilon, \bot)$ ($pc \in \{\texttt{tt}, \texttt{ff}\}$) in the `aSet`, then we can use $(pc, \epsilon, \bot)$ to update ψ_i in ψ. For example, if $op_i = \wedge$, indicating that all next states of s_i should satisfy

$$\begin{tikzcd}[column sep=tiny]
& \mathtt{t\_Obl}(\mathtt{E(tt\;U}\;p\mathtt{)}, s_0, \emptyset, \emptyset, A_1) \arrow[d, shift right] & & \\
& \mathtt{Obl}(\mathtt{E(tt\;U}\;p\mathtt{)}, s_0, \emptyset, \emptyset, A_2) \arrow[u, shift right, "{A_1 = A_2[(\mathtt{E(tt\;U}\;p\mathtt{)}, s_0) \mapsto (\mathtt{ff}, \epsilon, \bot)/ (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0) \mapsto (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0, \vee)]}"'] \arrow[d, shift right] & & \\
& \mathtt{t\_Obl}(p \vee \mathtt{EX(}\,\mathtt{E(tt\;U}\;p\mathtt{)}\,\mathtt{)}, s_0, (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0), \emptyset, A_2) \arrow[u, shift right] \arrow[d, shift right] & & \\
& \mathtt{Obl}(p \vee \mathtt{EX(}\,\mathtt{E(tt\;U}\;p\mathtt{)}\,\mathtt{)}, s_0, (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0), \emptyset, A_3) \arrow[u, shift right, "{A_2 = A_3 \cup \{(p \vee \mathtt{EX(}\,\mathtt{E(tt\;U}\;p\mathtt{)}\,\mathtt{)}, s_0) \mapsto (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0, \vee)\}}"'] \arrow[dl] \arrow[dr] \arrow[d, phantom, "\vee"] & & \\
\mathtt{t\_Obl}(p, s_0, (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0), \emptyset, A_4) \arrow[d, shift right] & & \mathtt{t\_Obl}(\mathtt{EX(}\,\mathtt{E(tt\;U}\;p\mathtt{)}\,\mathtt{)}, s_0, (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0), A_4, A_3) \arrow[d, shift right] & \\
\mathtt{Obl}(p, s_0, (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0), \emptyset, A_5) \arrow[u, shift right, "{A_4 = \{(p, s_0) \mapsto (\mathtt{ff}, \epsilon, \bot)\}}"'] \arrow[d, shift right] & & \mathtt{Obl}(\mathtt{EX(}\,\mathtt{E(tt\;U}\;p\mathtt{)}\,\mathtt{)}, s_0, (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0), A_4, A_6) \arrow[u, shift right, "{A_3 = A_6 \cup \{(\mathtt{EX(}\,\mathtt{E(tt\;U}\;p\mathtt{)}\,\mathtt{)}, s_0) \mapsto (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0, \vee)\}}"'] \arrow[dl] \arrow[dr] \arrow[d, phantom, "\vee"] & \\
\{\mathtt{ff}, \epsilon, \bot\} \arrow[u, shift right, "A_5 = \emptyset"'] & \mathtt{t\_Obl}(\mathtt{E(tt\;U}\;p\mathtt{)}, s_0, (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0), A_4, A_6) \arrow[d, shift right] & & (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0, \vee) \\
& \mathtt{Obl}(\mathtt{E(tt\;U}\;p\mathtt{)}, s_0, (\mathtt{E(tt\;U}\;p\mathtt{)}, s_0), A_4, A_7) \arrow[u, shift right, "{A_6 = A_7 \cup \{(\mathtt{E(tt\;U}\;p\mathtt{)}, s_0) \mapsto (\mathtt{ff}, \epsilon, \bot)\}}"'] \arrow[d, shift right] & & \\
& \{\mathtt{ff}, \epsilon, \bot\} \arrow[u, shift right, "A_7 = A_4"'] & &
\end{tikzcd}$$

Figure 5.3 Example of computing variation point obligations

φ_i, then in the case $pc = \mathtt{tt}$, ψ_i remains unaltered since the satisfiability of φ_i in one next state does not prove that φ_i is satisfied in all next states; on the other hand, if $pc = \mathtt{ff}$, then it can be concluded that the variation point obligation has not been satisfied at s_i.

Example. *Continuing our example, after variation point obligations for F_1, F_2 and F_3 are computed,* $\mathtt{aSet}$ $= \{(\varphi, s_0) \mapsto (\varphi, s_0, \vee)$, $(p \vee EX(\varphi), s_0) \mapsto (\varphi, s_0, \vee)$, $(p, s_0) \mapsto (ff, \epsilon, \bot)$, $(EX(\varphi), s_0) \mapsto (\varphi, s_0, \vee)$, $(\varphi, s_1) \mapsto (\varphi, s_1, \vee)$, $(p \vee EX(\varphi), s_1) \mapsto (\varphi, s_1, \vee)$, $(p, s_1) \mapsto (ff, \epsilon, \bot)$, $(EX(\varphi), s_1) \mapsto (\varphi, s_1, \vee)$, $(\varphi, s_2) \mapsto (ff, \epsilon, \bot)$, $(p \vee EX(\varphi), s_2) \mapsto (ff, \epsilon, \bot)$, $(p, s_2) \mapsto (ff, \epsilon, \bot)$, $(EX(\varphi), s_2) \mapsto (ff, \epsilon, \bot)\}$.
In the above $\mathtt{aSet}$, $(\varphi, s_0) \mapsto (\varphi, s_0, \vee)$ *and there exists* $s_0 \rightarrow s_2$ *in* $T^{F1,F3}$: $(\varphi, s_2) \mapsto (ff, \epsilon, \bot)$. *When performing* $\mathtt{update}$ *to the answer set, the second branch of the function is applied. This does not change the existing obligations in the* $\mathtt{aSet}$.

Algorithm 1 Analysis for Inter-Feature Loops

```
procedure INTERP($(\varphi, s)$, Dep)
  if $(\varphi, s) \mapsto (pc, \epsilon, \perp)$ return $pc$
  if $(\varphi, s) \in \texttt{Dep}$ then
    if $\varphi$ is gfp return tt else return ff
  end if
  $(\varphi, s) \mapsto (\varphi_1, s_1, op_1) op_2 \ldots op_{2k-2} (\varphi_k, s_k, op_{2k-1}) \in \texttt{aSet}$
  $\texttt{Next}_i := \bigcup_{s_i \to t_i} \{(\varphi_i, t_i) \mid s_i \to t_i \in T_c^{\texttt{FM}_m, \texttt{FM}_n}\}$
  for $1 \le i \le k$ do
    if $op_i = \wedge$ $\text{res}_i = \texttt{tt}$ else $\texttt{res}_i = \texttt{ff}$
    for $(\varphi_i, t_i) \in \texttt{Next}_i$ do
      $\texttt{res}_i = \texttt{res}_i\ op_i\ \text{INTERP}((\varphi_i, t_i), \texttt{Dep} \cup \{(\varphi, s)\})$
    end for
    if $i = 1$ $\texttt{res} = \texttt{res}_i$ else $\texttt{res} = \texttt{res}_{i-1}\ op_{2i-2}\ \texttt{res}_i$
  end for
  return res
end procedure
```

5.5.4 Step 3: Identifying Inter-Feature Loops from the Answer Set

To summarize, once the variation point obligations have been computed for all FMs (Steps 1 and 2) and for every $(\varphi, s) \mapsto \psi$ in aSet, each subformula (φ_i, s_i, op_i) of ψ has a corresponding $(\varphi_i, s_i) \mapsto \psi'$ in aSet, we can conclude that no further updates to aSet can be computed.

We can now search for any chain of variation point obligations from aSet to identify loops between features. An example of such a chain is the circular dependency between F_1 and F_2 in Fig 5.2: $(\varphi, s_0) \mapsto (\varphi, s_0, \vee)$ and $(\varphi, s_1) \mapsto (\varphi, s_1, \vee)$, where $s_0 \to s_1$ and $s_1 \to s_0$. The search for the inter-feature loops is done by applying the $\texttt{update}_\texttt{F}(\texttt{aSet})$ function on each element in aSet, where $\texttt{update}_\texttt{F}(\texttt{aSet})$ is defined as follows:

$$\texttt{update}_\texttt{F}(\texttt{aSet}) = \texttt{aSet}[(\varphi, s) \mapsto \psi / (\varphi, s) \mapsto (pc, \epsilon, \perp)]$$
$$\text{where } pc = \text{INTERP}((\varphi, s), \emptyset)$$

Algorithm 1 computes INTERP, taking as input parameters (φ, s) and the set Dep which records the elements on which the mapping result of (φ, s) depends. In Line 2, if $(\varphi, s) \mapsto (pc, \epsilon, \perp)$, then the result does not depend on other elements, and the procedure immediately returns pc. In this case the answer has been resolved. Otherwise (Lines 3–5), if the result depends on element in the set Dep, a circular dependency is identified, and the return result is computed based on whether or not φ is a greatest or a least fixed point formula (similar to the way we detected intra-feature loops during variation point obligation computation using the history set, see Fig. 5.1). If neither of the above, in Lines 6-15, the algorithm computes the dependency of results across features. Line 6 depicts the generic form of ψ in an answer set element $(\varphi, s) \mapsto \psi$ (according to Fig. 5.1). Then in Line 7, for each subformula ψ_i of ψ, where ψ_i=$(\varphi_i, s_i, op_{2i-1})$, t_i (the state connected to s_i) is identified and collected in the set $\texttt{Next}_i$. In the following lines (Line 10–12), INTERP is recursively invoked on each element of $\texttt{Next}_i$. The intermediate result $\texttt{res}_i$, with its default value set according to op_i being $\wedge$ or $\vee$ (Line 9), is aggregated with result computed for previous element(s) of $\texttt{Next}_i$ (Line 11). The final result to return, $\texttt{res}$, is computed by connecting all the intermediate $\texttt{res}_i$ using the same op_{2i-2} that connects all subformula ψ_i in ψ (Line 13).

Note that the above algorithm handles the situation where a variation point s_i has no state in another feature to connect to it. In such a situation, $\texttt{Next}_i$ is empty, and $\texttt{res}_i$ has its default value set as $\texttt{tt}$ or $\texttt{ff}$ according to op_i. Thus, any form of variation point obligation can be updated using INTERP and eventually be resolved to the final form of $(pc, \epsilon, \perp)$.

Example. *We perform* $\texttt{update}_F(\texttt{aSet})$ *on each element in the previously-computed* $\texttt{aSet}$ *(the sequence does not matter). For example, if the following element is picked first:* $(\varphi,s_1) \mapsto (\varphi,s_1,\vee)$*, we compute* INTERP$((\varphi,s_1),\emptyset)$ = INTERP$((\varphi,s_0),\ \{(\varphi,s_1)\})$ = INTERP$((\varphi,s_1),\ \{(\varphi,s_1),(\varphi,s_0)\})$ = ff*. We then replace* $(\varphi,s_1)\mapsto(\varphi,s_1,\vee)$ *with* $(\varphi,s_1)\mapsto (ff,\epsilon,\perp)$ *in the* $\texttt{aSet}$*. Other elements of* $\texttt{aSet}$ *are updated in a similar fashion until no change can be done to* $\texttt{aSet}$*. This means that all circular dependencies between these features have been resolved.*

5.5.5 Summary: Compositional Algorithm

Algorithm 2 presents the compositional algorithm introduced at the beginning of this section using the functions described above. This algorithm model checks the current product (a composition of features), reusing results from any previous model checking of those features for other products in the product line.

To summarize, given a composition $Comp_{seq}(\texttt{FM}_1, \texttt{FM}_2, \cdots, \texttt{FM}_m)$ and a formula φ, the algorithm first obtains the variation point obligations for $\texttt{FM}_1$ such that all its start states can

Algorithm 2 Compositional Model Checking

```
procedure COMPOSE($Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \cdots, \mathtt{FM}_m), \varphi$)
  $\mathtt{aSet}_{\mathtt{current}} := \emptyset$
  for each $s_0 \in S_0^1$ do
    $\mathtt{t\_Obl}(\varphi, s_0, \emptyset, \mathtt{aSet}_{\mathtt{current}}, \mathtt{aSet})$
    $\mathtt{aSet}_{\mathtt{current}} := \mathtt{aSet}$
    if $(\varphi, s_0) \mapsto (\mathtt{ff}, \epsilon, \bot) \in \mathtt{aSet}_{\mathtt{current}}$ return $\mathtt{aSet}_{\mathtt{current}}$
  end for
  repeat
    $tmp := \mathtt{aSet}_{\mathtt{current}}$
    for each $(\varphi', s') \mapsto (\varphi_1, s_1, op_1)op_2 \ldots op_{2k-2}(\varphi_k, s_k, op_{2k-1}) \in \mathtt{aSet}$
        $\wedge\ \varphi_i \notin \{\mathtt{tt}, \mathtt{ff}\} \wedge s_i \to t_i \in T_c^{\mathtt{FM}_m, \mathtt{FM}_n}$ do
      $\mathtt{t\_Obl}(\varphi_i, t_i, \emptyset, \mathtt{aSet}_{\mathtt{current}}, \mathtt{aSet})$
      $\mathtt{aSet}_{\mathtt{current}} := \mathtt{aSet}$
      $\mathtt{aSet}_{\mathtt{current}} := \mathtt{update}(\mathtt{aSet}_{\mathtt{current}})$
    end for
  until $(\mathtt{aSet}_{\mathtt{current}} = tmp)$
  return $\mathtt{update}_{\mathtt{F}}(\mathtt{aSet}_{\mathtt{current}})$
end procedure
```

satisfy φ (Line 3-7). From Lines 8 to 15, the variation point obligations of the other features connected to the variation points of $\mathtt{FM}_1$ are computed. The process of computing the variation point obligation is iterated until no more updates on the aSet can be made (Line 15). At this point, the function $\mathtt{update}_{\mathtt{F}}(\mathtt{aSet})$ is invoked to identify loops between features and infer results from variation point obligations represented in greatest and least fixed point formulas in the aSet. We say that $Comp_{seq}(\mathtt{FM}_1, \mathtt{FM}_2, \cdots, \mathtt{FM}_m) \models \varphi$ when for all start states s_0 of $\mathtt{FM}_1$, $((\varphi, s_0) \mapsto (\mathtt{tt}, \epsilon, \bot)) \in \mathtt{update}_{\mathtt{F}}(\mathtt{aSet})$.

Example. *So far, for example in Fig. 5.2, we have performed every step of Algorithm 2. Since $(\varphi, s_0) \mapsto (\mathtt{ff}, \epsilon, \bot) \in \mathtt{update}_{F}(\mathtt{aSet})$, the verification result is that $Comp_{seq}(F_1, F_2, F_3) \not\models \varphi$.*

5.6 Correctness Proof

In this section, we present the correctness proof of the COMPOSE algorithm described in Section 5.5.

Theorem 1 (Sound and Complete). *$Comp_{seq}(FM_1,FM_2,\cdots,FM_m) \models \varphi \Leftrightarrow \forall\ s_0 \in S_0^1$: $(\varphi,s_0) \mapsto (\mathtt{tt},\epsilon,\bot) \in$ COMPOSE($Comp_{seq}(FM_1,FM_2,\cdots,FM_m)$, φ).*

$$1.\ \frac{s \models^{\circ}_{H} \texttt{tt}\ [(\texttt{tt}, \epsilon, \bot)]}{\bullet} \qquad 2.\ \frac{s \models^{\circ}_{H} \texttt{ff}\ [(\texttt{ff}, \epsilon, \bot)]}{\bullet}$$

$$3.\ \frac{s \models^{\circ}_{H} p\ [(\texttt{tt}, \epsilon, \bot)]}{\bullet}\ p \in L(s) \qquad 4.\ \frac{s \models^{\circ}_{H} p\ [(\texttt{ff}, \epsilon, \bot)]}{\bullet}\ p \notin L(s)$$

$$5.\ \frac{s \models^{\circ}_{H} \neg\varphi\ [\neg\psi]}{s \models^{\circ}_{H} \varphi\ [\psi]} \qquad 6.\ \frac{s \models^{\circ}_{H} \varphi_1 \vee \varphi_2\ [\psi_1 \vee \psi_2]}{s \models^{\circ}_{H} \varphi_1\ [\psi_1] \quad s \models^{\circ}_{H} \varphi_2\ [\psi_2]}$$

$$7.\ \frac{s \models^{\circ}_{H} \texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2)\ [(\texttt{ff}, \epsilon, \bot)]}{\bullet}\ \{(\texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2), s) \in H\}$$

$$8.\ \frac{s \models^{\circ}_{H} \texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2)\ [\psi]}{s \models^{\circ}_{H \cup \{(\texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2), s)\}} \varphi_2 \vee (\varphi_1 \wedge \texttt{EX}(\texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2)))\ [\psi]} \{(\texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2), s) \notin H\}$$

$$9.\ \frac{s \models^{\circ}_{H} \texttt{EG}(\varphi)\ [(\texttt{tt}, \epsilon, \bot)]}{\bullet}\ \{(\texttt{EG}(\varphi), s) \in H\}$$

$$10.\ \frac{s \models^{\circ}_{H} \texttt{EG}(\varphi)\ [\psi]}{s \models^{\circ}_{H \cup \{(\texttt{EG}(\varphi), s)\}} \varphi \wedge \texttt{EX}(\texttt{EG}(\varphi)))\ [\psi]}\ \{(\texttt{EG}(\varphi), s) \notin H\}$$

$$11.\ \frac{s \models^{\circ}_{H} \texttt{EX}(\varphi)\ [\bigvee \psi_i]}{s_i \models^{\circ}_{H} \varphi\ [\psi_i]}\ \{s \to s_i, s \notin V\} \qquad 12.\ \frac{s \models^{\circ}_{H} \texttt{EX}(\varphi)\ [\bigvee \psi_i\ \vee\ (\varphi, s, \vee)]}{s_i \models^{\circ}_{H} \varphi\ [\psi_i]}\ \{s \to s_i, s \in V\}$$

Figure 5.4 Tableau rules for $\models^{\circ}_{H}$

Proof. The function COMPOSE($Comp_{seq}$(FM$_1$,FM$_2$,$\cdots$,FM$_m$), φ) returns aSet of variation point obligations (ψ) computed by iterative applications of t_Obl and update followed by INTERP. While t_Obl and update are used to compute the variation point obligations of different features and to compose them, INTERP computes the result of composition once all the features have been explored appropriately, i.e., no new variation point obligations can be obtained.

Therefore, the proof of correctness of COMPOSE algorithm can be realized by proving the correctness of t_Obl, update and INTERP functions.

• **Correctness of** t_Obl We introduce the notion of open-system verification to formalize variation point obligations. Recall that, a state s in FM satisfies CTL formula φ, denoted by $s \models \varphi$, if and only if s belongs to the semantics of φ. In the case of a FM with variation points where new features can be composed, we may not be able to infer whether $s \models \varphi$. Instead, we introduce a new predicate $\models^{\circ}_{H} \subseteq S \times \Phi \times \mathcal{H} \times \Psi$ where S is the set of states, Φ is the set of CTL formulas, $\mathcal{H}$ is the set of history sets and Ψ is the set of variation point obligation formulas. We write $s \models^{\circ}_{H} \varphi \; [\psi]$ to state that s satisfies φ in the context of a history H if and only if the interface obligation ψ is satisfiable. Following tableau-based local model checking technique Cleaveland (1990), we present in Fig. 5.4 the tableau rules for $\models^{\circ}_{H}$ predicate which captures the semantics of $\models^{\circ}_{H}$. In each rule of the form $\dfrac{a}{a_1, a_2, \ldots, a_n}$ denotes $a_1 \wedge a_2 \wedge \ldots \wedge a_n \Rightarrow a$.

Lemma 1. *$s \models^{o}_{\emptyset} \varphi \; [\psi] \Leftrightarrow (\psi \Leftrightarrow s \models \varphi)$.*

Proof. The above lemma holds trivially for the tableau rules 1–4. Rule 5 follows directly from the semantics of boolean connective $\Leftrightarrow$: $\neg\psi \Leftrightarrow s \models \neg\varphi$ is equivalent to $\psi \Leftrightarrow s \models \varphi$. Rule 6 corresponds to the disjunctive formula $\varphi_1 \vee \varphi_2$ where the variation point obligation of the disjunction is equal to the disjunction of the variation point obligations of the corresponding disjuncts φ_1 and φ_2.

Rules 7 and 8 handle the `EU` formula. The rules take into consideration the fixed point nature of the formula. $\texttt{E}(\varphi_1 \; \texttt{U} \; \varphi_2)$ is a least fixed point formula, the semantics of which is computed by least fixed point of the function: $f = \varphi_2 \;\vee\; (\varphi_1 \wedge EXf)$.

Note that proof of satisfiability of a least fixed point formula at a state cannot be obtained from a path with a loop (infinite path); least fixed point formulas always have a finite path proof. The history H keeps track of whether there is a loop in the proof of whether s satisfies $\texttt{E}(\varphi_1 \; \texttt{U} \; \varphi_2)$. If such a loop is detected, s does not satisfy $\texttt{E}(\varphi_1 \; \texttt{U} \; \varphi_2)$ in this loop and the interface obligation is $(\texttt{ff}, \epsilon, \bot)$. On the other hand, if a loop is not detected, the formula $\texttt{E}(\varphi_1 \; \texttt{U} \; \varphi_2)$ is expanded to its equivalent form and the history set is updated to include a new entry $(s, \texttt{E}(\varphi_1 \; \texttt{U} \; \varphi_2))$ to capture the fact that we have tried to identify the satisfiability conditions of $\texttt{E}(\varphi_1 \; \texttt{U} \; \varphi_2)$ at the state s.

Rules 9 and 10 deal with greatest fixed point formula $\texttt{EG}(\varphi)$. The semantics of $\texttt{EG}(\varphi)$ is computed from the greatest fixed point of the function: $f = \varphi \;\wedge\; EXf$.

Proof of satisfiability of the greatest fixed point formula at a state will contain loops (infinite paths). In this case, Rule 9 states that s satisfies $\texttt{EG}(\varphi)$ if the history set contains an entry

$(\texttt{EG}(\varphi), s)$, and the corresponding interface obligation is $(\texttt{tt}, \epsilon, \bot)$. The entry in H proves that there exists a path from s to itself where in each state in the path the formula φ is satisfied. This in turn proves that there exists an infinite path from s where every state satisfies $\texttt{EG}(\varphi)$. Rule 10 corresponds to the case where H does not include $(\texttt{EG}(\varphi), s)$, and as such the tableau rule expands the formula $\texttt{EG}(\varphi)$ to its equivalent form and updates the history set.

Finally, Rules 11 and 12 correspond to the case where the formula to be satisfied is of the form $\texttt{EX}(\varphi)$. If s is not a variation point (Rule 11), then the formula $\texttt{EX}(\varphi)$ can be satisfied if and only if φ is satisfied in any one of its next states. Therefore, the variation point obligation is computed from the *disjunction* of the variation point obligations corresponding to the satisfiability condition of φ at each next state. On the other hand, if s is a variation point (Rule 12), then the variation point obligation for s also includes the disjunction of $(\varphi, s, \vee)$. In short, if a feature, composed at the variation point s, satisfies φ then s satisfies $\texttt{EX}(\varphi)$.

The tableau rules consider all the syntactic constructs of CTL (constructs with universal path quantifiers can be obtained via negation and existential path formulas), and we have proved that all the rules are sound. The tableau constructed using these rules is always finite where the number of nodes in the tableau is bound by the number of states in the $\texttt{FM}$ and number of subformulas in φ. This concludes the proof of the lemma. □

Lemma 2. *$s \models^{o}_{H} \varphi\ [\psi] \Leftrightarrow (\varphi, s) \mapsto \psi \ \in\ \mathtt{aSet}_{out}$ where $\psi := \mathtt{t\_Obl}(\varphi, s, H, \emptyset, \mathtt{aSet}_{out})$.*

Proof. We show that the function `t_Obl` (Fig. 5.1) correctly implements the tableau rules in Fig. 5.4. Observe that, `t_Obl` invokes `Obl` in a mutually recursive fashion. Each rule for the definition of `Obl` directly encodes the tableau rules in Fig. 5.4. For example Rule 3 in Fig. 5.1 corresponds to Rules 3 and 4 in Fig. 5.4.

Note that the tableau rules allow repeated computation of $s \models^{o}_{H} \varphi\ [\psi]$ as they do not keep track of whether this computation has been completed in some other part of the tableau. Their implementation `t_Obl`, however, avoids such repeated (sub)computations, with the same parameters s, φ and H, using the fourth argument $\texttt{aSet}_{in}$ (case 1 in definition A for `t_Obl` in Fig. 5.4). The second case in definition A is little bit more involved. This is due to the fact that `aSet` is updated when among the recursive calls to `t_Obl` via `Obl` for the same s and φ, at least one of them is returned ahead of others. Therefore, the `aSet` needs to be updated after all such calls have returned. The updates result in disjunction or conjunction of each return depending on whether the formula under consideration is an existential or universal path property. In our setting, we only consider existential path properties which will be the result from the disjunction of all return valuations. This situation occurs when $\varphi := \mathtt{EX}(\varphi')$ and there is a loop on s. □

Proposition 1. $(\varphi, s) \mapsto (tt, \epsilon, \bot) \in$ *aSet* $\Leftrightarrow s \models \varphi$ *where* $(tt, \epsilon, \bot) := t\_Obl(\varphi, s, \emptyset, \emptyset, aSet)$.

Proof. The proposition follows from Lemma 1 and Lemma 2. □

• **Correctness of** `update`

Lemma 3. $(\varphi, s) \mapsto \psi \in update(aSet) \;\Leftrightarrow\; s \models^{o}_{\emptyset} \varphi\ [\psi]$.

Proof. Following the definition of `update`, there are two cases. In the first case, `update` does not alter the variation point obligation.

$$\begin{aligned}\begin{pmatrix}(\varphi, s) \mapsto \psi \in \mathtt{update}(\mathtt{aSet}) \wedge \\ \forall(\varphi', s', op) \in \mathtt{sub}(\psi) : \nexists(\varphi', t') \mapsto (pc, \epsilon, \bot) \in \mathtt{aSet}\end{pmatrix} &\Leftrightarrow \psi := \mathtt{t\_Obl}(\varphi, s, \emptyset, \emptyset, \mathtt{aSet}) \\ &\Leftrightarrow s \models^{o}_{\emptyset} \varphi\ [\psi] \text{ from Lemma 1}\end{aligned}$$

In the second case, where the variation point obligation is altered WLOG we will consider $\mathtt{FM}_1$

and $\texttt{FM}_2$ where s, s' are states in $\texttt{FM}_1$ and t' is a state in $\texttt{FM}_2$ such that $s' \to t' \in T_c^{\texttt{FM}_1,\texttt{FM}_2}$. Then,

$$\left(\begin{array}{l} (\varphi, s) \mapsto \psi[\psi'/\psi''] \in \texttt{update}(\texttt{aSet}) \wedge \\ \psi' = (\varphi', s', op) \ \wedge \ (\varphi', t') \mapsto (pc, \epsilon, \bot) \in \texttt{aSet} \\ \psi'' = \begin{cases} \psi' & \text{if } (pc = \texttt{tt}) \wedge (op_i = \wedge) \\ \psi' & \text{if } (pc = \texttt{ff}) \wedge (op_i = \vee) \\ (\texttt{tt}, \epsilon, \bot) & \text{if } (pc = \texttt{tt}) \wedge (op_i = \vee) \\ (\texttt{ff}, \epsilon, \bot) & \text{otherwise} \end{cases} \end{array}\right) \Leftrightarrow \left(\begin{array}{l} \psi = \texttt{t\_Obl}(\varphi, s, \emptyset, \emptyset, \texttt{aSet}_1) \wedge \\ (pc, \epsilon, \bot) = \\ \texttt{t\_Obl}(\varphi', t', \emptyset, \texttt{aSet}_1, \texttt{aSet}) \end{array}\right)$$

$$\begin{array}{ll} & \psi[\psi'/\psi''] = \texttt{t\_Obl}(\varphi, s, \emptyset, \emptyset, \texttt{aSet}_1) \\ \Leftrightarrow & \text{where } s \text{ is a state in} \\ & Comp_{seq}(\texttt{FM}_1, \texttt{FM}_2) \\ \Leftrightarrow & s \models_{\emptyset}^{\circ} \varphi \ [\psi[\psi'/\psi'']] \text{ from Lemma 1} \end{array}$$

□

- **Correctness of** $\texttt{INTERP}$

Lemma 4. *The maximum recursion depth of the function* $\texttt{INTERP}((\varphi, s), \emptyset)$ *is bound by* $|\texttt{aSet}|$.

Proof. Assume that $\texttt{INTERP}((\varphi, s), \emptyset)$ terminates at k recursion depth where $k > |\texttt{aSet}|$ where $|\texttt{aSet}|$ is the number of elements of the form $(\varphi_i, s_i) \mapsto \psi_i$. This implies that there are k calls to $\texttt{INTERP}$ with k different pairs (φ, s) which implies there are k elements in $|\texttt{aSet}|$. This leads to a contradiction proving that our assumption is incorrect. □

Lemma 5. *Given an* $\texttt{aSet}$ *such that* $\forall (\varphi, s) \mapsto \psi \in \texttt{aSet} \ \wedge \ \forall (\varphi', s', op') \in \texttt{sub}(\psi) \ \Rightarrow \ \forall s' \to t' \in T_c^{FM_m,FM_n} : (\varphi', t') \mapsto \psi' \ \in \ \texttt{aSet}$, *then the following holds:*

$$(\varphi, s) \mapsto \psi \in \texttt{aSet} \ \wedge \ s \models \varphi \Leftrightarrow \ \texttt{INTERP}((\varphi, s), \emptyset)$$

Proof. We first prove that INTERP handles inter-feature loops correctly. In case of loops, INTERP can be recursively invoked with the same pair (φ, s). Recall that satisfiability of greatest fixed point formulas (e.g., EG) require loops while satisfiability of least fixed point formulas (e.g. EU) by a state cannot be inferred from a loop. As such, INTERP returns true if the inter-feature loop represents a path corresponding to satisfiability of greatest fixed point; otherwise, it returns false.

For $(\varphi, s) \mapsto (pc, \epsilon, \bot) \in \texttt{aSet}$, the lemma is proved as follows:

$$\begin{aligned}
(\varphi, s) \mapsto (pc, \epsilon, \bot) \in \texttt{aSet} \;\wedge\; s \models \varphi \;\; &\Leftrightarrow \;\; s \models^{\circ}_{\emptyset} \varphi\ [(pc, \epsilon, \bot)] \;\wedge\; s \models \varphi \\
&\;\; \text{From Lemmas 2 and 3} \\
&\Leftrightarrow \;\; pc = \texttt{tt} \\
&\Leftrightarrow \;\; \texttt{INTERP}((\varphi, s), \emptyset)\ \texttt{INTERP}\text{ returns } pc \\
&\;\; \text{(see Line 2 in Algorithm 1)}
\end{aligned}$$

For $(\varphi, s) \mapsto \psi \in \texttt{aSet}$ where $\psi = (\varphi_1, s_1, op_1)op_2 \ldots op_{2k-2}(\varphi_k, s_k, op_{2k-1}) \in \texttt{aSet}$ the proof proceeds as follow:

$$\begin{aligned}
&((\varphi, s) \mapsto (\varphi_1, s_1, op_1)op_2 \ldots op_{2k-2}(\varphi_k, s_k, op_{2k-1}) \in \texttt{aSet} \wedge \;\; s \models \varphi \\
&\Leftrightarrow ((\varphi_1, s_1, op_1)op_2 \ldots op_{2k-2}(\varphi_k, s_k, op_{2k-1}) \;\; \Leftrightarrow \;\; s \models \varphi) \;\; \wedge \;\; s \models \varphi \\
&\text{From Lemma 1} \\
&\Leftrightarrow \texttt{INTERP}((\varphi, s), \emptyset) \text{ using Lines 7–14 in Algorithm 1}
\end{aligned}$$

□

Finally, Theorem 1 can be proved as follows. Lines 2–7 in Algorithm 2 compute the variation point obligation of $\texttt{FM}_1$ in the sequence. Lines 8–15 compute the variation point obligation for all $\texttt{FM}_i$ $(1 \leq i \leq m)$. At each iteration the $\texttt{aSet}_{current}$ obtained at the end satisfies the following:

$$(\varphi, s) \mapsto \psi \in \texttt{aSet}_{current} \;\; \Leftrightarrow \;\; s \models^{\circ}_{\emptyset} \varphi\ [\psi] \Leftrightarrow (\psi \Leftrightarrow s \models \varphi)$$

The above follows from Lemmas 1, 2 and 3.

The terminating condition of the iteration $\texttt{aSet}_{current} = tmp$ is satisfied only when no new additions and updates are possible in the answer set. That is, $\forall (\varphi, s) \mapsto \psi \in \texttt{aSet}_{current} \;\; \wedge$

$\forall(\varphi', s', op') \in \texttt{sub}(\psi) \Rightarrow \forall s' \rightarrow t' : (\varphi', t') \mapsto \psi' \in \texttt{aSet}_{current}$. Finally, at Line 16 the algorithm `COMPOSE` returns the result correctly inferring whether or not $Comp_{seq}(\texttt{FM}_1, \texttt{FM}_2, \ldots, \texttt{FM}_m) \models \varphi$. This follows from Lemmas 4 and 5.

□

This concludes the proof that the `COMPOSE` algorithm is sound and complete.

5.7 Implementation

We have implemented the compositional model checking algorithm in a research prototype model checker. In the implementation, the *history set* is implemented by recording the set of states visited along each path during the computation of obligations inside a single feature. The *answer set* is implemented by associating an *obligation table* at every visited state, with each entry of the table recording an incoming property for that state and the computed obligation for the next state, e.g., $((\varphi, s_i), (\varphi', s_j, op))$ where $s_i \rightarrow s_j$.

Virtual-state mechanism. Since a variation point may have transitions to states inside the same feature ("intra-feature next state"), or to entry states of other feature(s) ("inter-feature next state"), we need to store obligations for both types of next states in a variation point state's *obligation table*. In order to store the two types of information and to tell them apart in a convenient manner, we introduce a *virtual state* construct associated with each variation point, to record the obligations each variation point has for its future next state(s) in another feature. (A *virtual state* is not an existing state in a feature. Rather, it represents the "virtual" next state of a variation point.) This way, the *obligation table* for a variation point state appears the same as for any other states, while the *obligation table* for a *virtual state* appears in the following form: $((\varphi, s_v), (\varphi, s_v, op))$. Thus, $\texttt{t\_Obl}(\varphi, s_v, H, \texttt{aSet}_{in}, \texttt{aSet}_{out})$ for a *virtual state* will return (φ, s_v, op) and terminate.

When an obligation computation for a feature has completed, each state's *obligation table* is updated with the resulting variation point obligation (represented as virtual-state obligation in the implementation). This mechanism supports any state in becoming a variation point. Moreover, obligations for other features can be easily retrieved from the virtual states' constraint tables.

Reuse of variation point obligations. In the product-line development, different products share the common features (e.g., the Base Controller feature in the pacemaker example). In order to be able to reuse the model checking results, we need to keep different versions of the answer set in order to avoid inadvertently changing the obligations generated at those common features for the different compositions. For example, in the pacemaker product line, checking the RateResponsivePacemaker requires checking the composition of the base controller feature and the rate-responsive extension feature. This will apply the `update` function on the `aSet`, resolving the obligations generated at the variation points of the base controller. (By "resolving" it we mean that those obligations no longer depend on other features, but have a definite `tt` or `ff` result instead). Those resolved obligations at the base controller cannot be used to check its composition with the mode-transitive extension in the ModeTransitivePacemaker. To solve this problem, we keep a different copy of the answer set for the mode-transitive composition.

In the implementation, we allow the computed variation point obligations for one feature to be reused in different feature compositions. This is achieved by creating *updated obligation tables* at the entry states (i.e., initial states as defined in Def. 1) of each feature to store the updated obligation after feature composition. For example, for an entry state, the original *obligation table* may have an entry of the form $((\varphi, s_0), (\varphi', s_i, op))$, while the *updated obligation table* may have an entry of the form $((\varphi, s_0), (\texttt{ff}, \epsilon, \perp))$. Entries in the original *obligation table* are reusable as they do not depend on other features, while entries in the *updated obligation table* are composition-dependent. By distinguishing versions of *obligation tables* at the entry states, other non-entry states' *obligation tables* will not be updated by the composition, and thus can be safely reused.

Moreover, different features may connect to the same feature `FM` through overlapping but non-identical sets of variation points. Thus, for the model checking results for `FM` to be reused, we need to specify the union of the sets of variation points for different feature compositions. This will lead to unused or disconnected variation points for a specific feature composition.

These variation points are *closed* by automatically updating their corresponding obligations with $(\texttt{ff}, \epsilon, \perp)$ or $(\texttt{tt}, \epsilon, \perp)$ depending on whether a universal or existential path formula is satisfied at that point (as described in the $\texttt{update}_{\texttt{F}}(\texttt{aSet})$ function).

The support for reuse here allows the model checking to be conducted in an incremental fashion, and reduces the rechecking effort.

Optimization of compositional model checking. Since the same model checking algorithm will be applied repeatedly for different products in a product line, optimization of the algorithm can achieve more savings than in the single systems. In this work, we optimize the compositional algorithm by providing the following "short-cuts":

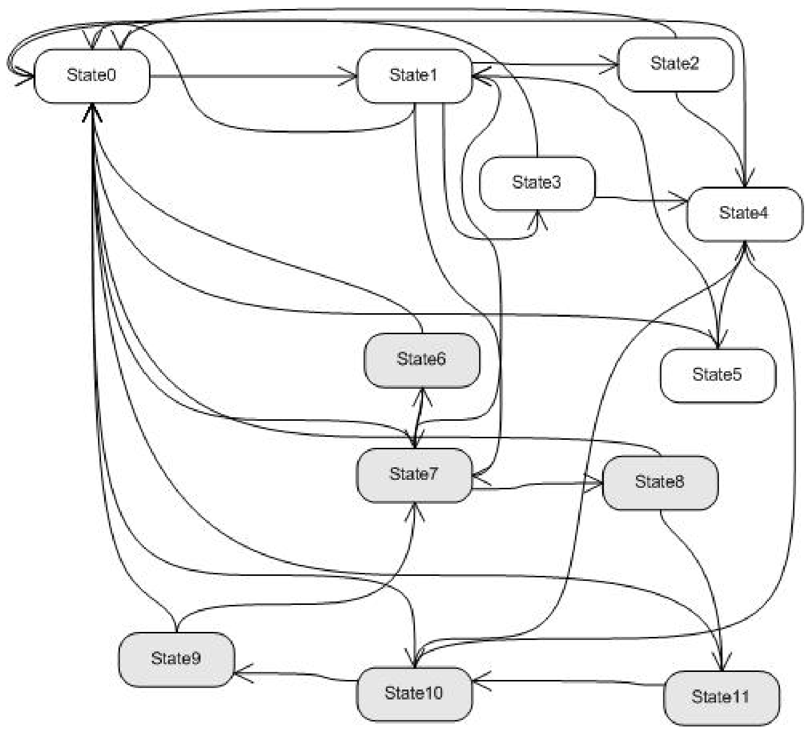
1. if a disjunct (resp. conjunct) evaluates to `tt` (resp. `ff`), the remaining disjuncts (resp. conjuncts) in a disjunctive (conjunctive) formula are not analyzed.
2. when one variation point of `FM` has connected to multiple states from other features, after each such connection, a check is run to determine whether that variation point's obligation has been updated to $(\texttt{ff}, \epsilon, \perp)$ or $(\texttt{tt}, \epsilon, \perp)$ in the *updated obligation tables* at the entry states of `FM`. If it has, the algorithm stops computing obligations for the subsequent connections of that variation point.

These optimizations further reduce the amount of checking needed to generate the final result for each product-line composition.

5.8 Case Study

We evaluated our technique by conducting experiments on the pacemaker product line discussed in Section 5.3. Figures 5.5 and 5.6 depict how the Mode Transitive Extension ($\texttt{FM}_1$) and Rate Responsive Extension ($\texttt{FM}_2$) are sequentially composed with the BasePacemaker's controller ($\texttt{FM}_0$). The states in the extensions are shown in grey. The variation points are the states which have outgoing transitions leading to another model in the composition. The propositions satisfied at each state are shown in the corresponding tables below each figure.

In the controller for the fourth product (not shown), the ModeTransitive-RateResponsive Pacemaker, $\texttt{FM}_1$ and $\texttt{FM}_2$ are sequentially composed with $\texttt{FM}_0$, as in Figures 5.5 and 5.6. $\texttt{FM}_1$



State	True Propositions	State	True Propositions
s_0	timerOff=1	s_6	timerSenseTimeUp=1; sensorOn=1; triggeredMode=1;
s_1	sensorOn=1; inhibitedMode=1;		
s_2	sensed=1; sensorOn=1; inhibitedMode=1;	s_7	sensorOn=1; triggeredMode=1;
		s_8	sensed=1; sensorOn=1; triggeredMode=1;
s_3	timerSenseTimeUp=1; pulseGen=1; inhibitedMode=1;	s_9	timerRefractoryTimeUp=1; sensorOn=1; triggeredMode=1;
s_4	inhibitedMode=1;		
s_5	timerRefractoryTimeUp=1; sensorOn=1; inhibitedMode=1;	s_{10}	triggeredMode=1;
		s_{11}	pulseGen=1; triggeredMode=1;

Figure 5.5 Base Controller and Mode-Transitive Extension

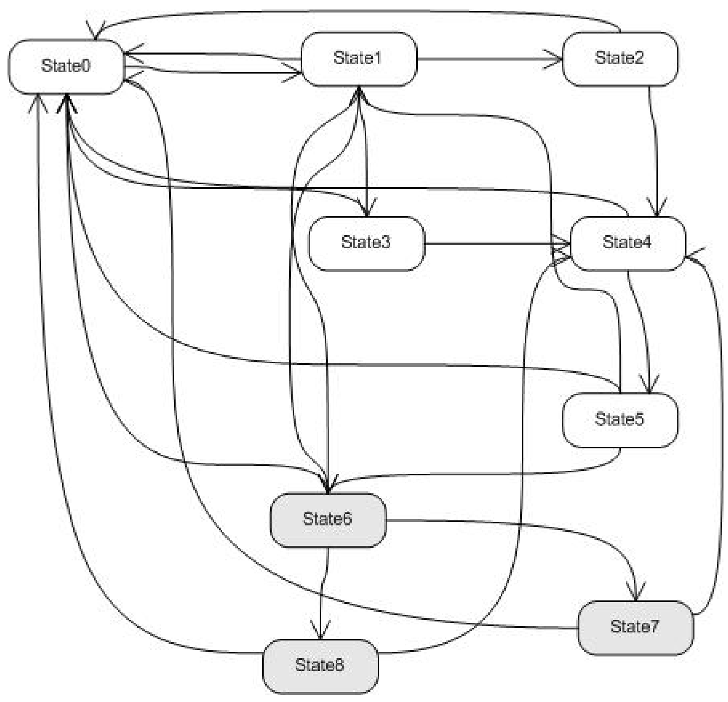
and $\mathtt{FM}_2$ do not connect directly one to the other since a pacemaker controller can be in either the Triggered Mode or Inhibited Mode, but not in both at the same time.

The following CTL formula describes the required property for the product line that was textually introduced in Section 5.3:

$$\mathtt{AG}\big((\mathtt{sensed=0} \wedge \mathtt{timerSenseTimeUp=1} \wedge \mathtt{inhibitedMode=1}) \Rightarrow$$
$$\mathtt{EF}(\mathtt{pulseGen=1} \wedge \mathtt{inhibitedMode=1})\big)$$

This formula was used in the following evaluation.

To evaluate the space and time performance of our approach, we compared our composi-



State	True Propositions
s_6	sensorOn=1; inhibitedMode=1; upperRateLimit=1;
s_7	sensorOn=1; sensed=1; inhibitedMode=1; upperRateLimit=1;
s_8	timerShortSenseTimeUp=1; pulseGen=1; inhibitedMode=1; upperRateLimit=1;

Figure 5.6 Base Controller and Rate-Responsive Controller Extension

tional model checking (CMC) technique with non-compositional model checking (NMC) for the four products in the product line and recorded the experimental results in Table 5.1. In the table, MT denotes ModeTransitive, while RR denotes RateResponsive. The variables (`t_Obl`) and (`Obl`) record the number of times the `t_Obl` and `Obl` functions are performed, respectively. Similarly, ($\texttt{Obl}_{\texttt{top\_test}}$) and ($\texttt{Obl}_{\texttt{test}}$) record the times these two assistant functions in our implementation have been invoked. The assistant functions serve to conduct lightweight tests (by "lightweight" we mean that they check only the current state), e.g., to see if one branch of a disjunct formula is true during obligation computation. The variable ($\texttt{add}_{\texttt{obl}}$) records the number of times an obligation is added to our implementation of `aSet`. Finally, # of state visits records the number of times the states in a model are visited.

Table 5.1 Test data for pacemaker product line

	Non-Compositional Model Checking (NMC)				**Compositional Model Checking (CMC)**			
# of Invocations	Base Pace-maker	MT Pace-maker	RR Pace-maker	MT-RR Pace-maker	Base Pace-maker	MT Pace-maker	RR Pace-maker	MT-RR Pace-maker
(`t_Obl`)	33	68	52	87	40	51	30	28
(`Obl`)	20	38	29	47	22	22	12	5
($\mathtt{Obl}_{\mathtt{top\_test}}$)	65	125	94	154	71	76	39	20
($\mathtt{Obl}_{\mathtt{test}}$)	64	124	93	153	66	65	32	6
($\mathtt{add}_{\mathtt{Obl}}$)	91	179	135	222	102	126	72	61
# of State Visits	13	30	23	40	16	23	13	14

The results of NMC are obtained from checking each of the four products in its entirety, without breaking it into separate features. No variation points are specified, and the start state is always resolved to a true or false result at the end of the checking.

The results of CMC are obtained from calculating the test data for the added features and connections in each product. For example, verifying the BasePacemaker involves checking $\mathtt{FM}_0$ with states 1, 4, and 5 as its variation points (i.e., the union set of the variation points needed by the MT and RR extensions), plus applying the $\mathtt{update}_{\mathtt{F}}(\mathtt{aSet})$ to get the final result. We found that, as expected, generating obligations at the variation points and performing $\mathtt{update}_{\mathtt{F}}(\mathtt{aSet})$ introduced a slight overhead for the base product that was amortized over its subsequent reuse. Verifying the RateResponsivePacemaker involves checking the $\mathtt{FM}_2$ with states 6, 7, and 8 as its variation points, plus checking the connections from $\mathtt{FM}_2$ to $\mathtt{FM}_0$, and applying the `update` and $\mathtt{update}_{\mathtt{F}}(\mathtt{aSet})$ functions. Test results were similarly collected for the other two products.

Table 5.1 shows that the compositional model checking approach does provide savings in the product line. For example, in NMC the cost for checking the product line (measured in (`t_Obl`)) was 33+68+52+87=240, while the cost in CMC was 40+51+30+28=149. This is because the common features (e.g., $\mathtt{FM}_0$) were checked repeatedly in NMC. If no prior checking for any of the features had been done, the cost for checking the RateResponsivePacemaker (measured in (`t_Obl`)) would have been 40+30=70, which is more than the value of 52 for the NMC. This difference is due to the cost of generating assets for reuse, i.e., generating obligations at the variation points and maintaining a different aSet (for applying the update and $\texttt{update}_\texttt{F}$(aSet) functions) for each composition. To summarize, as with the product-line approach itself, CMC shows savings when features are reused.

5.9 Discussion

In this section, we discuss the usefulness and the applicability of this technique in different systems, in different types of changes, and in different phases of product-line development.

5.9.1 Handling Parallel Composition of Features

Although this work targets sequential composition of features (Sec. 5.4.1), the technique described in this chapter can handle many instances of parallel composition of features. Specifically, it can handle the situation in which a Feature A is composed in parallel with a Feature B, Feature A provides only control signals to B, and the CTL property to be verified does not involve A. In such a case, the control signals from A can be modeled as abstract events in B. In other words, these abstract events represent the effect of feature A on feature B.

Recall that in sequential composition, the set of transitions of the composed model is the union of the transitions from the composed features (local transitions). This means that a transition in the composed model (a global transition) is just one of the local transitions. However, in parallel composition, a global transition is a combination of several local transitions that may occur at the same time Huth and Ryan (2004).

For an example of parallel composition, consider the interaction of Extra Sensor feature (the functionality of the extra sensor component) and the RateResponsiveExtension feature in the RateResponsivePacemaker. These two features are composed in parallel because both feature models can make a transition at the same time. Since the CTL property to be verified here does not depend on the behavior of the Extra Sensor, we model the Extra Sensor's effect on the extension to the base controller's functionality by introducing an abstract event in the extension ("upperRateLimit=1" in Figure 5.6(b)'s Table).

Another example of parallel composition that is handled by our technique is the Timer feature (whose behavior is parallel to that of the controller's, not shown in Fig. 2.1). We again handle the effect of the Timer by introducing abstract events (i.e., "timerSenseTimeUp=1", "timerRefractoryTimeUp=1" and "timerShortSenseTimeUp=1" in Figure 5.6(b)'s Table) in the controller's model. This allows us to handle the rest of the feature compositions in a pure sequential fashion.

Abstracting such events is currently done manually. In the future, work on generating assumptions from the environment, e.g., [Giannakopoulou et al. (2002); Gheorghiu et al. (2007)] could possibly be incorporated to handle such an abstraction process in an automatic fashion.

5.9.2 Suitability for Product-line Evolution

During product-line evolution, structural changes to a composed feature model may occur, e.g., adding a new feature, replacing or changing an existing feature, or deleting a feature. The model checking technique described in this chapter supports evolution by minimizing the amount of re-checking that is required.

Adding a new feature through a pre-specified variation point s_i means that one or several states are introduced–$s_i \rightarrow t_i \in T_c^{\mathtt{FM}_m,\mathtt{FM}_n}$ (Line 10 of Algorithm 2). Thus, t_Obl(φ_i, t_i, $\emptyset$, $\mathtt{aSet}_{\mathtt{current}}$, aSet) is invoked for each such t_i, together with each of the formula φ_i that appears in the obligation of s_i (Line 11 of Algorithm 2). This change does not require re-checking other features because we can have the t_Obl function update the answer set obtained from the previous check of the feature composition. The subsequent update and $\mathtt{update}_F$ function will be performed on the updated answer set.

On the other hand, adding a new feature through an ad-hoc variation point (i.e., not specified in the checked model prior to the addition) is more complicated as there will not have been any obligations previously generated for such an ad-hoc variation point. An example of this in the pacemaker example is when variation points s_1 and s_5 are specified for the base controller feature to check the composition with the RateResponsiveExtension. Later when it is composed with the ModeTransitiveExtension, an extra variation point s_4 needs to be added, but no variation point obligation has yet been generated for s_4 in the prior checking of the base controller feature.

To take advantage of previous model checking, for such an ad-hoc variation point s, we identify all elements in the answer set from the previous check that have the form $(\varphi, s) \mapsto \psi$. For each such element, rechecking φ at s using the function $\texttt{t\_Obl}$ (with s a variation point) will generate the variation point obligations needed to check the added new feature. The rest of the process for model checking an added feature is the same as the above situation. Rechecking φ is very light-weight as all the states like s', $s' \to s \in \texttt{FM}_m$ have been checked already, so applying $\texttt{t\_Obl}$ on those states can readily reuse the existing elements like $(\varphi', s') \mapsto \psi'$ in the answer set.

The process for handling other changes in the composed feature model (e.g., replacing or changing an existing feature, or deleting a feature) is similar to the above description. However, since the affected features in this case may already be connected to other features, updates to variation point obligations of the affected changed features may result in rechecking all features directly and indirectly connected to them ((Line 10–15 of Algorithm 2). (Indirect connections refer to the situation in which some features do not have transitions to the changed feature but are connected to those features that have such transitions.) The important point is that features that are not affected will not be rechecked.

When structural changes to a feature occur (e.g., new transitions and states are added, or existing transitions and states are modified), we can divide the original feature into several "mini feature"s, representing the isolated changed parts, and the original model sans those changed parts, respectively. Rechecking for the original model can thus be carried out in a way similar to sequentially composing the changed parts with previously checked parts. This allows us to reuse as much of the still valid answer set elements from the previous checking as possible, and to check if some of the original obligations are preserved after the change (by "preserved" it means that the corresponding answer set elements involving those obligations do not need to be updated).

In summary, variation point obligations provide the modularity needed to model check product lines. The support for any state in a model to become a variation point, as well as the storage of previous obligation-generation results in the answer set, help reduce the rechecking load in case of changes. Because product lines routinely experience significant change over their lifetimes, the continued usefulness of previous model checking results to the product line development contributes to the practicality of this technique.

5.9.3 Consistency with Product-line Development

Product-line engineering is typically partitioned into two phases: Domain Engineering and Application Engineering Weiss and Lai (1999). A product line is initially defined by its common and variable features 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. The technique described in this chapter incorporates model checking into both phases of product-line development.

Domain engineering phase. The feature behavioral models constructed for the product-line features, the properties specified for the product line, as well as the variation point obligations generated for each of the common features regarding those shared properties, are among the reusable assets created in this phase. If model checking finds any error in the models, the design, the requirements, or the properties that capture the requirements, updates are made to solve those errors and those updates re-verified before advancing to the next phase.

Application engineering phase. Reusing the product-line assets previously created, different feature compositions are specified in this phase for different products. Thus, the obligations generated in the domain-engineering phase are reused in the application-engineering phase to conduct model checking of new products. The verification results, as well as the checked features, are among the product-specific assets created in this phase. As described in Section 5.9.2, these assets are maintained and updated during product-line evolution.

Incorporating model checking into both phases not only helps reduce the number of verification runs needed to ensure that a new product satisfies a common product-line property, but also allows the model checking effort to be conducted in an incremental fashion. In the future, we plan to integrate the work described here with a prototype tool we have developed Liu et al. (2008b), to better manage the models, as well as the properties and verification results in a product-line setting, and to help enhance the traceability of the assets generated in both phases.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

This chapter summarizes work presented in this dissertation, followed by a description of the contributions and future directions.

6.1 Summary

Product-line engineering presents an advantageous approach to developing software systems because the reuse can reduce the development time and cost. Yet, handling the interactions among variant features in the product-line setting is more complex than in traditional software systems because addition or 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 dissertation presented a safety analysis technique for software product lines using model-based development and model checking. In the front end, this analysis combined product-line software fault tree analysis (SFTA) and state-based modeling of critical components to identify potentially unsafe feature interactions. In the backend, this analysis employed model checking on the feature behavioral models to check against the property specifications in order to provide enhanced assurance on the interactions among features. The formal feature behavioral models and property specifications were derived from the state models and safety requirements, respectively. Both the model checking and the property specification process have been adapted to recognize the product-line variation mechanisms, so that results for individual variations in features or properties can be appropriately combined to reason about the whole product.

The results of the front end analysis guided and provided useful information for the formal verification in the backend. Exercising the state models made evident which parts relate to a certain safety property. Since the formal models could be derived from the state models relevant to that safety property, the scope needed for formal verification could be reduced. Similarly, the temporal logic properties were made more readily identified through the forbidden scenario and required scenario derivation. Problems found in formal verification could then also be traced back to concrete scenarios in order to update the design using scenario execution as described in Chapter 3. This integration of model-based development and model checking thus provided enhanced assurance and focus on safety-related concerns.

6.2 Contributions

The work described here provides a reusable safety analysis framework for safety-critical product lines. It enhances existing safety analysis by providing:

1. A safety-analysis guided, model-based approach that provides guidelines for users to construct the behavioral model of a product line's significant, safety-related variations and to check whether the variations and the behavior they introduce jeopardize the safety properties (Chapter 3). This method shows: (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.

2. A tool-supported technique that guides users in structured reuse of property specifications for model-checking the members of a product line (Chapter 4). Properties specified via the tool are traceable to the underlying product-line requirements, the SMV models, and the verification results. The tool enables reuse of shared product-line properties, as well as of product-line-specific patterns of properties, while carefully preserving any distinctions among

the product-line members. Results from application to the two product lines show that the tool also can manage the changes and re-verification needed as the product line evolves (e.g., as new members or features are added). By making it easier to specify, manage, and reuse properties, it helps make model-checking of product-line systems more practical.

3. An incremental and compositional model checking technique for performing sequential composition of different features in a product-line setting (Chapter 5). The resulting composition was shown to be sound and complete. This technique generates obligations at the variation points such that the feature composition satisfies the desired property if and only if the features added at variation points satisfy the corresponding obligations. By computing and managing variation point obligations, we enable reuse of previous verification results when a new product is composed. Re-checking occurs only when and as needed.

Additionally, this approach removes restrictions on how features can be sequentially composed, providing more flexibility in how features interact than existing techniques and bringing models more in line with real-world product lines. The technique accommodates product-line evolution by identifying obligations at these new variation points from previous obligations computed at those points. Evaluation done using a prototype implementation to model check a simplified pacemaker product line shows that this technique can reduce the amount of re-verification needed to assure that a required property holds for each new product in the product line.

6.3 Future Work

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. Several future research directions are outlined below:

Integrate with existing automated safety analysis tools

One of the drawbacks of FTA and FMEA is that they have to be done manually, thus primarily depending on the personnel who conduct the analysis. However, because of their easy-to-interpret forms and wide acceptance, some recent safety analysis techniques have been established to automatically generate results in those forms. For example, Joshi et al. (2007) generate FTA from AADL models with injected faults. Grunske et al. (2005b) generate FMEA through model checking the formal model of behavioral trees with component failure modes injected. These automations, when properly integrated, could greatly enhance the efficiency and scalability of our approach.

Tackle Composition Diversity

The compositional model checking algorithm currently targets sequential compositional of features. Section 5.9 has described how it can also handle certain instances of parallel composition of features, i.e., when the property does not involve any feature that solely provides control signals to other feature(s) in the parallel composition. These control signals may be able to be abstracted via generating assumptions from the environment (e.g., [Giannakopoulou et al. (2002); Gheorghiu et al. (2007)]) to provide better support for parallel compositions. Furthermore, interfaces for sequential and parallel compositions may be able to be unified to provide adaptive composition mechanisms, so that different types of modular composition can be handled.

Extend Configuration Management Techniques

The Formula Editor tool presented in Chapter 4 allows reuse of model checking results as evidence to prove that each product in the product line satisfies its safety requirements. This management facility could be extended to include a richer set of product-line assets such as the original requirements, the state models, and the interface obligations. This way, these assets can be better traced and maintained throughout the product-line development and evolution. This also allows the front end and backend safety analysis to be more smoothly integrated.

BIBLIOGRAPHY

Abadi, M. and Lamport, L. (1995). Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535.

Ardis, M. A. and Cuka, D. A. (1999). Defining families—commonality analysis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 671–672, Los Alamitos, CA, USA. IEEE Computer Society Press.

Armoni, R., Fix, L., Flaisher, A., Grumberg, O., Piterman, N., Tiemeyer, A., and Vardi, M. Y. (2003). Enhanced vacuity detection in linear temporal logic. In *CAV*, pages 368–380.

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., and Zettel, J. (2002). *Component-based product line engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Autili, M., Inverardi, P., and Pelliccione, P. (2006). A scenario based notation for specifying temporal properties. In *SCESM '06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 21–28, New York, NY, USA. ACM.

Basu, S. and Ramakrishnan, C. R. (2006). Compositional analysis for verification of parameterized systems. *Theoretical Computer Science*, 354(2):211–229.

Batory, D., Benavides, D., and Ruiz-Cortes, A. (2006). Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47.

Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., and Rodeh, Y. (2001). The temporal logic sugar. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 363–367, London, UK. Springer-Verlag.

Bennett, K. H. and Rajlich, V. T. (2000). Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA. ACM.

Blundell, C., Fisler, K., Krishnamurthi, S., and Hentenryck, P. V. (2004). Parameterized interfaces for open system verification of product lines. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 258–267, Washington, DC, USA. IEEE Computer Society.

Bosch, J. (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Bosch, J. (2004). Software variability management. *Science of Computer Programming*, 53(3):255–258.

Campbell, L. A., Cheng, B. H. C., McUmber, W. E., and Stirewalt, K. (2002). Automatically detecting and visualising errors in uml diagrams. *Requir. Eng.*, 7(4):264–287.

Childs, A., Greenwald, J., Jung, G., Hoosier, M., and Hatcliff, J. (2006). Calm and cadena: Metamodeling for component-based product-line development. *Computer*, 39(2):42.

Clarke, E. M., Grumberg, O., and Peled, D. A. (2000). *Model Checking*. MIT Press.

Clauß, M. (2001). Modeling variability with uml. In online proceedings of the GCSE 2001 Young Researchers Workshop. Retrieved 27 Sept. 2008. URL=http://www.info.uni-karlsruhe.de/ heuzer/GCSE-YRW2001/program.html.

Cleaveland, R. (1990). Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–748.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Păsăreanu, C. S., Robby, and Zheng, H. (2000). Bandera: extracting finite-state models from java source code. In *ICSE '00:*

Proceedings of the 22nd international conference on Software engineering, pages 439–448, New York, NY, USA. ACM.

Czerny, B. and Heimdahl, M. (1998). Automated integrative analysis of state-based requirements. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, page 125, Washington, DC, USA. IEEE Computer Society.

Dehlinger, J. and Lutz, R. R. (2004). Software fault tree analysis for product lines. In *HASE*, pages 12–21.

Dehlinger, J. and Lutz, R. R. (2005). A product-line requirements approach to safe reuse in multi-agent systems. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7.

Dehlinger, J. and Lutz, R. R. (2006). Plfaultcat: A product-line software fault tree analysis tool. *Automated Software Engg.*, 13(1):169–193.

Deng, G., Lenz, G., and Schmidt, D. (2005). Addressing domain evolution challenges in model-driven software product-line architectures. In *Proceedings of the ACE/IEEE MODELS 2005 Workshop on Model-Driven Development for Software Product Lines: Fact or Fiction?*

Doerr, J. (2002). *Requirements Engineering for Product Lines: Guidelines for Inspecting Domain Model Relationships*. PhD dissertation, University of Kaiserslautern.

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

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA. IEEE Computer Society Press.

Ellenbogen, K. A. and Wood, M. A. (2005). *Cardiac Pacing and ICDs (fourth edition)*. Blackwell Publishing, Inc.

Fantechi, A. and Gnesi, S. (2007). A behavioural model for product families. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 521–524, New York, NY, USA. ACM.

Feng, Q. and Lutz, R. R. (2005). Bi-directional safety analysis of product lines. *J. Syst. Softw.*, 78(2):111–127.

Fischbein, D., Uchitel, S., and Braberman, V. (2006). A foundation for behavioural conformance in software product line architectures. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 39–48, New York, NY, USA. ACM.

Geppert, B., Li, J. J., Rößler, F., and Weiss, D. M. (2004). Towards generating acceptance tests for product lines. In *ICSR*, pages 35–48.

Geppert, B., Mockus, A., and Rößler, F. (2005). Refactoring for changeability: A way to go? In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*, page 13, Washington, DC, USA. IEEE Computer Society.

Geppert, B. and Rößler, F. (2004). Effects of refactoring legacy protocol implementations: A case study. In *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium*, pages 14–25, Washington, DC, USA. IEEE Computer Society.

Gheorghiu, M., Giannakopoulou, D., and Păsăreanu, C. S. (2007). Refining interface alphabets for compositional verification. In *TACAS*, pages 292–307.

Giannakopoulou, D., Păsăreanu, C. S., and Barringer, H. (2002). Assumption generation for software component verification. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA. IEEE Computer Society.

Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

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

Grunske, L., Kaiser, B., and Reussner, R. (2005a). Specification and evaluation of safety properties in a component-based software engineering process. *Component-based Software Development For Embedded Systems: An Overview of Current Research Trends*, 3778:249 – 274.

Grunske, L., Lindsay, P. A., Yatapanage, N., and Winter, K. (2005b). An automated failure mode and effect analysis based on high-level design specification with behavior trees. In *IFM*, pages 129–149.

Hanks, K. S., Knight, J. C., and Holloway, C. M. (2002). The role of natural language in accident investigation and reporting guidelines. In *Proceedings of the 2002 Workshop on the Investigation and Reporting of Incidents and Accidents, Glasgow, Scotland, July, 2002)*. URL=citeseer.ist.psu.edu/hanks02role.html.

Harel, D. and Marelly, R. (2003). *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine (1st edition)*. Springer.

Harms-Ringdahl, L. (2001). *Safety Analysis: Principles and Practice in Occupational Safety (second edition)*. CRC Press, New York, NY, USA.

Havelund, K., Lowry, M. R., and Penix, J. (2001). Formal analysis of a space-craft controller using SPIN. *Software Engineering*, 27(8):1000–9999.

Holt, A. (1999). Formal verification with natural language specifications: guidelines, experiments and lessons so far. *South African Computer Journal*, 24:253–257.

Huth, M. and Ryan, M. (2004). *Logic in Computer Science: modelling and reasoning about systems (second edition)*. Cambridge University Press.

Jacobson, I., Griss, M., and Jonsson, P. (1997). *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley.

Jörges, S., Margaria, T., and Steffen, B. (2006). Formulabuilder: a tool for graph-based modelling and generation of formulae. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 815–818, New York, NY, USA. ACM.

Joshi, A., Vestal, S., and Binns, P. (2007). Automatic generation of static fault trees from aadl models. In *In DSN Workshop on Architecting Dependable Systems, Edinburgh, Scotland - UK, June 2007*.

Kaivola, R. (2005). Formal verification of pentium 4 components with symbolic simulation and inductive invariants. In Etessami, K. and Rajamani, S. K., editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 170–184. Springer.

Kang, K. (2006). Software product line research topics. The Product Line Research Panel, 10th International Software Product Line Conference (SPLC 2006), Baltimore, USA, 2006. Retrieved 27 Sept. 2008. URL=http://www.sei.cmu.edu/splc2006/SPLC06-ResearchPanel-KK.pdf.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168.

Kang, K. C., Lee, J., and Donohoe, P. (2002). Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65.

Kishi, T. and Noda, N. (2006). Formal verification and software product lines. *Commun. ACM*, 49(12):73–77.

Konrad, S. and Cheng, B. H. C. (2005). Facilitating the construction of specification pattern-based properties. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 329–338, Washington, DC, USA. IEEE Computer Society.

Kurshan, R. P. (2004). Evolution of model checking into the eda industry. In *ATVA*, pages 2–6.

Lauenroth, K. and Pohl, K. (2007). Towards automated consistency checks of product line requirements specifications. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 373–376, New York, NY, USA. ACM.

Leveson, N. G. (1995). *Safeware: system safety and computers*. ACM, New York, NY, USA.

Li, H. C., Krishnamurthi, S., and Fisler, K. (2005). Modular verification of open features using three-valued model checking. *Automated Software Engg.*, 12(3):349–382.

Littlewood, B. and Strigini, L. (1993). Validation of ultrahigh dependability for software-based systems. *Commun. ACM*, 36(11):69–80.

Liu, J. (2007). Handling safety-related feature interaction in safety-critical product lines. In *ICSE Doctoral Symposium '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 85–86, Washington, DC, USA. IEEE Computer Society.

Liu, J., Basu, S., and Lutz, R. (2008a). Generating variation-point obligations for compositional model checking of software product lines. Technical report, 08-04, Computer Science, Iowa State University.

Liu, J., Dehlinger, J., and Lutz, R. (2005a). Safety analysis of software product lines using state-based modeling. In *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 21–30, Washington, DC, USA. IEEE Computer Society.

Liu, J., Dehlinger, J., and Lutz, R. (2007a). Safety analysis of software product lines using state-based modeling. *Journal of Systems and Software*, 80(11):1879–1892.

Liu, J., Dehlinger, J., Sun, H., and Lutz, R. (2007b). State-based modeling to support the evolution and maintenance of safety-critical software product lines. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 596–608, Washington, DC, USA. IEEE Computer Society.

Liu, J., Hauptman, M., Lutz, R., Geppert, B., and Rößler, F. (2008b). A tool-supported technique for specification & management of model-checking properties for software product lines. Technical report, 08-05, Computer Science, Iowa State University.

Liu, J., Lutz, R., and Rajan, H. (2007c). The role of aspects in modeling product line variabilities. In *Proceedings of the First Workshop on Aspect-Oriented Product Line Engineering (AOPLE), Portland, Oregon, October, 2006.* Retrieved 27 Sept. 2008. URL=http://www.cs.iastate.edu/˜lss/publications/Liu_Lutz_Rajan_AOPLE1.pdf.

Liu, J., Lutz, R. R., and Thompson, J. M. (2005b). Mapping concern space to software architecture: a connector-based approach. In *ICSE Workshop on the Modeling and Analysis of Concerns in Software (MACS)*, pages 1–5.

Loer, K. and Harrison, M. D. (2006). An integrated framework for the analysis of dependable interactive systems (ifadis): Its tool support and evaluation. *Automated Software Engg.*, 13(4):469–496.

Lu, D. and Lutz, R. R. (2002). Fault contribution trees for product families. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 231, Washington, DC, USA. IEEE Computer Society.

Lutz, R. R. (2000a). Extending the product family approach to support safe reuse. *J. Syst. Softw.*, 53(3):207–217.

Lutz, R. R. (2000b). Software engineering for safety: a roadmap. In *ICSE - Future of SE Track*, pages 213–226.

McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA.

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

Model Checking Group at CMU (2006). The smv system. Model Checking @CMU. Carnegie Mellon University. Retrieved 27 Sept. 2008. URL=http://www.cs.cmu.edu/ modelcheck-/smv.html.

Mondragon, O., Gates, A. Q., and Roach, S. (2003). Prospec: Support for elicitation and formal specification of software properties. *Electr. Notes Theor. Comput. Sci.*, 89(2).

Northrop, L. and Clements, P. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Northrop, L. and McGregor, J. (First Quater 2003). Components as products. News@SEI. Ed. Janet Rex. Carnegie Mellon University. Retrieved 27 Sept. 2008. URL=http://www.sei.cmu.edu/news-at-sei/columns/software-product-lines/2003/1q03/software-product-lines-1q03.htm.

Ossher, H. and Tarr, P. (1999). Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717)16APR99, IBM T.J. Watson Research Center.

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

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer.

Prehofer, C. (2004). Plug-and-play composition of features and feature interactions with statechart diagrams. *Software and System Modeling*, 3(3):221–234.

Reps, T. and Teitelbaum, T. (1984). The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48.

Rhapsody (2005). Rhapsody tutorial for rhapsody in c. Release 6.0 MR-1. I-Logix, Inc.

Robby, Dwyer, M. B., and Hatcliff, J. (2006). Bogor: A flexible framework for creating software model checkers. In *TAIC-PART '06: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 3–22, Washington, DC, USA. IEEE Computer Society.

Rushby, J. (1995). Formal methods and their role in the certification of critical systems. Technical Report CSL-95-01, Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA.

Schmid, K. and Verlage, M. (2002). The economic impact of product line adoption and evolution. *IEEE Softw.*, 19(4):50–57.

Schwanke, R. W. and Lutz, R. R. (2004). Experience with the architectural design of a modest product family. *Softw. Pract. Exper.*, 34(13):1273–1296.

Smith, M. H., Holzmann, G. J., and Etessami, K. (2001). Events and constraints: A graphical editor for capturing logic requirements of programs. In *RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 14–22, Washington, DC, USA. IEEE Computer Society.

Smith, R. L., Avrunin, G. S., Clarke, L. A., and Osterweil, L. J. (2002). Propel: an approach supporting property elucidation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 11–21, New York, NY, USA. ACM.

Svahnberg, M. and Bosch, J. (1999). Characterizing evolution in product-line architectures. In *Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications (SEA'99)*.

Svahnberg, M., van Gurp, J., and Bosch, J. (2005). A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754.

Szyperski, C. (1998). *Component software: beyond object-oriented programming.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Thang, N. T. (2005). *Incremental Verification of Consistency in Feature-Oriented Software.* PhD dissertation, Japan Advanced Institute of Science and Technology.

Van Langenhove, S. and Hoogewijs, A. (2004). Integrating cadence smv in the verification of uml software. In *Proceedings of the Dutch Proof Tools Days*, pages 15–29, Nijmegen.

Vito, B. D. (2000). High-automation proofs for properties of requirements models. *International Journal on Software Tools for Technology Transfer*, 3(1):20–31.

Wang, X. (2005). A modular model checking algorithm for cyclic feature compositions. Master's thesis, Worcester Polytechnic Institute.

Webber, D. L. and Gomaa, H. (2004). Modeling variability in software product lines with the variation point model. *Sci. Comput. Program.*, 53(3):305–331.

Weiss, D. M. and Lai, R. (1999). *Software Product Line Engineering: A Family-Based Software Development Process.* Addison-Wesley.

Xie, F. and Browne, J. C. (2003). Verified systems by composition from verified components. *SIGSOFT Softw. Eng. Notes*, 28(5):277–286.

Yuan, Y. and Detlor, B. (2005). Intelligent mobile crisis response systems. *Commun. ACM*, 48(2):95–98.

Zave, P. (1993). Feature interactions and formal specifications in telecommunications. *Computer*, 26(8):20–29.