Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estratégias Comutativas para Análise de Confiabilidade em Linha de Produtos de Software

Thiago Mael de Castro

Brasília
2016

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estratégias Comutativas para Análise de Confiabilidade em Linha de Produtos de Software

Thiago Mael de Castro

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Vander Ramos Alves

Coorientador
Prof. Dr. Leopoldo Motta Teixeira

Brasília
2016

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

CC355e

Castro, Thiago Mael de
Estratégias comutativas para análise de confiabilidade em linha de produtos de software / Thiago Mael de Castro; orientador Vander Ramos Alves; co-orientador Leopoldo Motta Teixeira. -- Brasília, 2016.
105 p.

Dissertação (Mestrado - Mestrado em Informática) - Universidade de Brasília, 2016.

1. Linhas de produtos de software. 2. Análise de linha de produtos. 3. Análise de confiabilidade. 4. Model checking. 5. Verificação. I. Alves, Vander Ramos, orient. II. Teixeira, Leopoldo Motta, co orient. III. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estratégias Comutativas para Análise de Confiabilidade em Linha de Produtos de Software

Thiago Mael de Castro

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof. Dr. Vander Ramos Alves (Orientador)
CIC/UnB

Prof. Dr. Alexandre Cabral Mota
CIn/UFPE

Prof. Dr. Mauricio Ayala-Rincón
CIC/UnB

Prof.ª Dr.ª Célia Ghedini Ralha
Coordenadora do Mestrado em Informática

Brasília, 18 de novembro de 2016

Dedicatória

Aos bonobos.

Agradecimentos

À Isabela, minha amada esposa e amiga, sem a qual este trabalho não teria sido possível. Obrigado por compreender os momentos em que eu estive ausente e por me fazer parar de trabalhar, vez ou outra, para refrescar a cabeça.

Aos meus pais, Valéria e Gilvan, e ao meu irmão, Miguel, pela confiança que sempre depositaram em mim. Obrigado, especialmente, por me ensinarem a buscar respostas por mim mesmo e me motivarem, constantemente, a superar minhas próprias limitações.

Ao meu orientador, Prof. Vander Alves, pela inestimável orientação ao longo do desenvolvimento deste trabalho e por me apresentar ao fantástico mundo das linhas de produtos. Obrigado por me estimular a explorar idéias interessantes, mesmo quando eu não via luz no fim do túnel. Estendo os agradecimentos ao meu coorientador, Prof. Leopoldo Teixeira, pelas discussões sempre produtivas. Mais do que orientar, vocês participaram ativamente da pesquisa e me proporcionaram um aprendizado sem tamanho.

Ao colega de orientação, André Lanna, por ter idealizado a ReAna e ter semeado a idéia de nossa linha de produtos de ferramentas de análise. Obrigado por compartilhar sua experiência acadêmica e suas dicas musicais.

Aos professores Pierre-Yves Schobbens e Sven Apel, pelas revisões rigorosas e pelo feedback preciso. Suas contribuições para a consistência deste trabalho foram inestimáveis. Estendo os agradecimentos aos membros da banca examinadora, professores Alexandre Mota e Maurício Ayala-Rincón, pelas sugestões e críticas que contribuíram para o aprimoramento do trabalho.

Ao Exército Brasileiro, pelas oportunidades ímpares de crescimento profissional. Agradeço, em particular: ao TC Alisson, amigo e mentor na engenharia de software; ao Cel Kohl, pela confiança, pela paciência e por sempre advogar em prol da realização deste trabalho; ao Exmo. Sr. Gen Bráulio, por visualizar a importância do aperfeiçoamento acadêmico de seus subordinados e confiar que eu me tornaria um “bom mestre”; ao Ricardo, que tentou durante muitos anos me convencer a prosseguir nos estudos acadêmicos (até que conseguiu); e a todos os demais amigos da Divisão de Comando e Controle, pela compreensão nos meus momentos de ausência e pela excelência profissional—vocês me inspiram a ser um engenheiro melhor a cada *quantum* de tempo arbitrariamente definido.

"It is hard to claim that you know what you are doing,
unless you can present your act as a deliberate choice
out of a possible set of things you could have done as well."
(Edsger W. Dijkstra, On Program Families)

Resumo

Engenharia de linha de produtos de software é uma forma de gerenciar sistematicamente a variabilidade e a comunalidade em sistemas de software, possibilitando a síntese automática de programas relacionados (*produtos*) a partir de um conjunto de artefatos reutilizáveis. No entanto, o número de produtos em uma linha de produtos de software pode crescer exponencialmente em função de seu número de características, tornando inviável verificar a qualidade de cada um desses produtos isoladamente.

Existem diversas abordagens *cientes de variabilidade* para análise de linha de produtos, as quais adaptam técnicas de análise de produtos isolados para lidar com a variabilidade de forma eficiente. Tais abordagens podem ser classificadas em três dimensões de análise (*product-based*, *family-based* e *feature-based*), mas, particularmente no contexto de análise de confiabilidade, não existe uma teoria que compreenda (a) uma especificação formal das três dimensões e das estratégias de análise resultantes e (b) prova de que tais análises são equivalentes uma à outra. A falta de uma teoria com essas propriedades impede que se raciocine formalmente sobre o relacionamento entre as dimensões de análise e técnicas de análise derivadas, limitando a confiança nos resultados correspondentes a elas.

Para preencher essa lacuna, apresentamos uma linha de produtos que implementa cinco abordagens para análise de confiabilidade de linhas de produtos. Encontrou-se evidência empírica de que as cinco abordagens são equivalentes, no sentido em que resultam em confiabilidades iguais ao analisar uma mesma linha de produtos. Além disso, formalizamos três das estratégias implementadas e provamos que elas são corretas, contanto que a abordagem probabilística para análise de confiabilidade de produtos individuais também o seja. Por fim, apresentamos um diagrama comutativo de passos intermediários de análise, o qual relaciona estratégias diferentes e permite reusar demonstrações de corretude entre elas.

Palavras-chave: Linhas de produtos de software, Análise de linha de produtos, Análise de confiabilidade, Model checking, Verificação

Abstract

Software product line engineering is a means to systematically manage variability and commonality in software systems, enabling the automated synthesis of related programs (*products*) from a set of reusable assets. However, the number of products in a software product line may grow exponentially with the number of features, so it is practically infeasible to quality-check each of these products in isolation.

There is a number of *variability-aware* approaches to product-line analysis that adapt single-product analysis techniques to cope with variability in an efficient way. Such approaches can be classified along three analysis dimensions (product-based, family-based, and feature-based), but, particularly in the context of reliability analysis, there is no theory comprising both (a) a formal specification of the three dimensions and resulting analysis strategies and (b) proof that such analyses are equivalent to one another. The lack of such a theory prevents formal reasoning on the relationship between the analysis dimensions and derived analysis techniques, thereby limiting the confidence in the corresponding results.

To fill this gap, we present a product line that implements five approaches to reliability analysis of product lines. We have found empirical evidence that all five approaches are equivalent, in the sense that they yield equal reliabilities from analyzing a given product line. We also formalize three of the implemented strategies and prove that they are sound with respect to the probabilistic approach to reliability analysis of a single product. Furthermore, we present a commuting diagram of intermediate analysis steps, which relates different strategies and enables the reuse of soundness proofs between them.

Keywords: Software product lines, Product-line analysis, Reliability analysis, Model checking, Verification

List of Figures

2.1 Feature model of the BSN product line [Rodrigues et al., 2015]. 9
2.2 Example graph view of a DTMC and the corresponding reachability probability. 15
2.3 Example graph view of a PMC and the intuition for the corresponding reachability probability expression. 18
2.4 Elimination of state s in the parametric reachability probability algorithm (adapted from Hahn et al. [2011]). 19
2.5 ADD A_f representing the Boolean function f in Equation (2.2). 20
2.6 Example of an arithmetic operation over ADDs. 21
2.7 Example of an ITE operation over ADDs. 22
2.8 Alternative ordering for encoding the Boolean function f in Equation (2.2) as an ADD. 22

3.1 Outline of the research phases. Magnifying glasses denote analysis tools, and turnstiles ($\vdash$) denote theories. 24
3.2 Example of reliability-annotated sequence diagram used in a product line. The prob tags annotate each message with its reliability, and opt fragments mark variation points with corresponding presence conditions as guards. . 27
3.3 Example of PMCs obtained from the sequence diagram in Figure 3.2. Dashed green arrows denote the dependency relation. 28
3.4 Outline of the implemented feature-family-based analysis approach. At the right side, we denote the resulting ADDs: sets of green circles (each denoting a configuration) mapped to sets of yellow squares (each denoting the corresponding reliability values—“products”). 30
3.5 Outline of the implemented feature-product-based analysis approach. At the right side, we denote the resulting reliabilities by yellow squares. The green box denotes iteration over all valid configurations $c \in [\![FM]\!]$. 32
3.6 Outline of the implemented product-based analysis approach. Nested squares represent DTMCs that result from composition. 33

3.7 Outline of the implemented family-based analysis approach. Nested squares represent variability-encoded PMCs, with the outermost square denoting the result from embedding the variability of the whole product line into the root PMC. At the extreme right, we see a single ADD that maps all valid configurations to their respective reliabilities (cf. Figure 3.4). 34
3.8 Outline of the implemented family-product-based analysis approach. In contrast with Figure 3.7, the result is represented by a reliability value (yellow box), independently computed for each valid configuration. 35
3.9 Outline of the relation between the implemented product-line reliability analysis strategies. 36
3.10 Feature model of the reliability analysis software product line (ReAna-SPL). 37
3.11 Comparison of derivation and variability encoding. Green solid arrows denote new transitions, and dashed red arrows represent transitions that are removed by the transformation process. 43
3.12 Feature model of the reliability analysis software product line (ReAna-SPL) after feedback from the formalization phase (Chapter 4). 44

4.1 Vending machine product line example. 47
4.2 Annotative PMC for the vending machine. 48
4.3 Compositional PMCs for the vending machine. 53
4.4 Example of a partial composition of PMCs. 55
4.5 Trivial compositional PMC $\widetilde{\mathcal{P}}$. 55
4.6 Example compositions for the vending machine. 56
4.7 Commutative diagram of product-line reliability analysis strategies. 57
4.8 Example of family-product-based analysis (α_v followed by σ) in contrast to a product-based analysis (λ followed by α) of an annotative PMC. 61
4.9 Statement of Theorem 1. 62
4.10 Example of lifted expression evaluation using $\hat{p}$. 67
4.11 Statement of Theorem 3. 68
4.12 Statement of Theorem 4. 69

A.1 Complete annotative PMC for the vending machine. 86
A.2 Compositional PMCs for the vending machine. 87

B.1 Overall theory structure. 89
B.2 Dependencies for Theorem 1 (Soundness of family-product-based analysis). 90
B.3 Dependencies for Theorem 4 (Soundness of family-based analysis). 91

List of Tables

3.1 Correspondence between valid configurations, analysis strategies, and analysis steps. 38
3.2 Product lines used for empirical validation. 40
3.3 Maximum relative errors for each analysis strategy, using the product-based analysis as a baseline. 40
3.4 Occurrences of the *if-then-else* pattern. 42
3.5 Correspondence between valid configurations, analysis strategies, and analysis steps after formalization of strategies. 44

List of Definitions

1 Property (Reachability probability for DTMCs) . . . 16
1 Definition (Parametric Markov Chain) . . . 16
2 Definition (Expression evaluation) . . . 16
3 Definition (Well-defined evaluation) . . . 17

4 Definition (Annotative PMC) . . . 49
5 Definition (Presence function) . . . 49
6 Definition (Evaluation factory) . . . 50
7 Definition (Annotative probabilistic model) . . . 50
8 Definition (DTMC derivation) . . . 51
9 Definition (Non-parametric model checking) . . . 58
10 Definition (Product-based analysis of annotative models) . . . 58
11 Definition (Parametric model checking) . . . 59
12 Definition (Expression evaluation) . . . 60
13 Definition (Family-product-based analysis) . . . 60
14 Definition (Expression lifting) . . . 63
15 Definition (Lifted evaluation factory) . . . 65
16 Definition (Variability-aware expression evaluation) . . . 65
17 Definition (Family-based analysis) . . . 68

List of Theorems and Lemmas

1 Lemma (Parametric probabilistic reachability soundness) 19

2 Lemma (Evaluation well-definedness for annotative models) 51
3 Lemma (Commutativity of PMC and expression evaluations) 61
1 Theorem (Soundness of family-product-based analysis) 61
4 Lemma (Soundness of expression lifting) 63
2 Theorem (Soundness of variability-aware expression evaluation) 66
5 Lemma (Soundness of lifted annotative evaluation factory) 66
3 Theorem (Soundness of expression evaluation using $\hat{p}$) 67
4 Theorem (Soundness of family-based analysis) 68

Acronyms

ADD Algebraic Decision Diagram.

CTL Computation Tree Logic.

DTMC Discrete-Time Markov Chain.

PCTL Probabilistic Computation Tree Logic.

PMC Parametric Markov Chain.

SPL Software Product Line.

Contents

List of Figures ix

List of Tables xi

List of Definitions xii

List of Theorems and Lemmas xiii

Acronyms xiv

1 Introdução 1
1.1 Definição do Problema 2
1.2 Solução Proposta 3
1.3 Resumo das Contribuições 3
1.4 Estrutura 4

2 Background 6
2.1 Software Product Lines 6
2.1.1 Main Concepts 7
2.1.2 Adoption Strategies 10
2.1.3 Variability Implementation 10
2.1.4 Product-Line Analysis 12
2.2 Reliability Analysis 14
2.2.1 Parametric Markov Chains 16
2.2.2 Parametric Probabilistic Reachability 18
2.3 Algebraic Decision Diagrams 20

3 A Product Line of Product-line Analysis Tools 23
3.1 Research Method 23
3.1.1 Threats to Validity 25
3.2 Domain Engineering 26

3.2.1 Overview . . . 26
3.2.2 Product-line Extraction . . . 31
3.2.3 Reactive Evolution . . . 32
3.3 Product Line of Product-line Reliability Analysis Tools . . . 35
3.3.1 Quality Assessment . . . 38
3.3.2 Empirical Validation . . . 39
3.4 Theory Development . . . 41

4 Commuting Strategies for Product-line Reliability Analysis 46
4.1 Markov-chain Models of Product Lines . . . 46
4.1.1 Annotative Models . . . 48
4.1.2 Compositional Models . . . 52
4.2 Reliability Analysis Strategies . . . 55
4.2.1 Product-based Strategy . . . 58
4.2.2 Family-based Strategies . . . 59

5 Conclusion 70
5.1 Related Work . . . 71
5.2 Future Work . . . 75

Bibliography 77

A Probabilistic Models 85

B Theory Dependencies 88

Capítulo 1

Introdução

Engenharia de linha de produtos de software é uma forma de gerenciar sistematicamente a variabilidade e a comunalidade em sistemas de software, possibilitando a síntese automatizada de programas relacionados (conhecidos como *variantes* ou *produtos*) a partir de um conjunto de artefatos reutilizáveis (*artefatos de domínio*) [Apel et al., 2013a; Clements and Northrop, 2001; Pohl et al., 2005]. Em uma linha de produtos, a variabilidade é modelada em termos de *features*, que são características perceptíveis e relevantes para algum interessado no sistema (*stakeholder*) [Czarnecki and Eisenecker, 2000]. Essa metodologia melhora a produtividade e o tempo de colocação no mercado (*time-to-market*), além de facilitar a personalização em massa de software [Pohl et al., 2005].

Linhas de produtos vêm sendo amplamente utilizadas, tanto industrial [van der Linden et al., 2007; Weiss, 2008] quanto academicamente [Apel et al., 2013a; Clements and Northrop, 2001; Heradio et al., 2016; Pohl et al., 2005], particularmente em sistemas críticos [Domis et al., 2015; Dordowsky et al., 2011; Lanman et al., 2013; Rodrigues et al., 2015; Weiss, 2008]. *Model checking* é uma técnica particularmente interessante para garantia de qualidade desse tipo de sistemas. Essa técnica de verificação explora todos os estados possíveis de um modelo do sistema de forma sistemática, verificando que tal modelo satisfaz determinada propriedade [Baier and Katoen, 2008].

O número de produtos em uma linha de produtos pode crescer exponencialmente em função do número de *features*, dando origem a uma *explosão combinatória* do espaço de configurações [Apel et al., 2013a; Bodden et al., 2013; Classen et al., 2010, 2011]. Dessa forma, é frequentemente inviável verificar a qualidade de cada um dos produtos isoladamente. Não obstante, técnicas de verificação de software para o caso de produtos isolados são largamente utilizadas industrialmente, o que torna interessante explorar sua maturidade para aumentar a qualidade ao mesmo tempo em que se reduz custos e riscos [Baier and Katoen, 2008].

Existem diversas abordagens para análise de linha de produtos que adaptam técnicas de análise consagradas de forma a lidar com variabilidade [Thüm et al., 2014]. Em particular, diversas técnicas de *model checking* foram alçadas à operação em linhas de produtos [Chrszon et al., 2016; Classen et al., 2013, 2011, 2014; Dubslaff et al., 2015; Ghezzi and Molzam Sharifloo, 2013; Kowal et al., 2015; Nunes et al., 2012; Rodrigues et al., 2015; Thüm et al., 2014]. Dentre essas técnicas, o presente trabalho se concentra em análise de confiabilidade, que é a verificação de uma propriedade de existência probabilística [Grunske, 2008] e pode ser intuitivamente vista como a probabilidade de que um sistema não falhe.

1.1 Definição do Problema

Análises de linhas de produtos podem ser classificadas em três dimensões: *product-based* (a análise é realizada sobre produtos ou sobre modelos destes), *family-based* (apenas artefatos de domínio e suas combinações válidas são verificados) e *feature-based* (artefatos de domínio que implementam uma dada *feature* são analisados isoladamente, independente de suas combinações válidas com outros artefatos) [Thüm et al., 2014]. Mais de uma dimensão pode ser explorada em uma dada técnica, dando origem a estratégias híbridas — como *feature-family-based* e *family-product-based*, por exemplo. No entanto, abordagens existentes para o problema de alçar análises de software consagradas a linhas de produtos normalmente se concentram na dimensão *family-based* [Chrszon et al., 2016; Dubslaff et al., 2015; Midtgaard et al., 2015; von Rhein et al., 2016], relacionando-a somente com a dimensão *product-based* para argumentar sobre corretude. No contexto de análise de confiabilidade, em especial, não existe teoria que compreenda (a) uma especificação formal das três dimensões e estratégias de análise resultantes e (b) demonstrações de que tais análises são equivalentes umas às outras.

A falta de uma teoria com essas características impede que se raciocine formalmente sobre a relação entre as dimensões e análises derivadas, limitando a confiança nos resultados de análise correspondentes. De fato, demonstrar que um método de análise produz um resultado correto é uma preocupação fundamental, especialmente para a verificação de sistemas críticos. Ademais, um profissional da indústria deve ser capaz de selecionar uma estratégia de análise de acordo com o problema em questão, baseado nos compromissos assumidos em termos de espaço e tempo [Thüm et al., 2014]. Enquanto não houver evidência de que estratégias diferentes são mutuamente equivalentes, estudos empíricos que as comparem terão resultados com validade limitada.

1.2 Solução Proposta

Com base na taxonomia de análise de linhas de produtos proposta por Thüm et al. [2014], foram investigadas cinco abordagens para análise de confiabilidade de linhas de produtos: uma *product-based*, uma *family-based*, uma *family-product-based*, uma *feature-family-based* e uma *feature-product-based*. A solução proposta apresenta dois aspectos: (a) as cinco abordagens foram implementadas como uma linha de produtos de ferramentas para análise de linhas de produtos, e o processo de engenharia de domínio que levou a essa linha de produtos foi documentado; e (b) três das estratégias de análise implementadas foram formalizadas, tendo sido demonstrado que são equivalentes uma à outra — o que estabelece sua corretude e a relação entre elas.

O processo começou com a implementação de uma ferramenta para análise de confiabilidade segundo uma estratégia *feature-family-based*, originada a partir de trabalhos relacionados internamente ao grupo de pesquisa. A partir daí, foram aplicadas estratégias extrativas e reativas para adoção de linha de produtos, de forma a iniciar e evoluir uma linha de produtos de ferramentas de análise. Foi realizada uma comparação empírica das estratégias de análise implementadas, a partir da qual encontrou-se evidência de que elas encontram confiabilidades iguais para uma mesma linha de produtos.

Usando a implementação resultante e o conhecimento de domínio adquirido através de sua construção, formalizamos explicitamente as abordagens *product-based*, *family-based* e *family-product-based*. Além disso, foram identificadas quatro abordagens alternativas ao longo desse processo de formalização: uma *product-based*, uma *family-based*, e duas *family-product-based*. A formalização das estratégias *feature-family-based* e *feature-product-based* (ambas implementadas), assim como das estratégias alternativas identificadas, é objeto de pesquisa em andamento.

Provamos que as estratégias de análise formalizadas são corretas, contanto que a abordagem probabilística para análise de confiabilidade de produtos individuais também o seja. Ademais, apresentamos um diagrama comutativo dos passos de análise intermediários, o qual relaciona estratégias diferentes e permite o reúso de provas de corretude entre elas. Nesse sentido, reforçamos a evidência de que a aplicação de qualquer das estratégias formalizadas produz o mesmo resultado.

1.3 Resumo das Contribuições

As principais contribuições deste trabalho são as seguintes:

- Formalização de três estratégias para análise de confiabilidade de linhas de produtos de software, em acordo com a classificação proposta por Thüm et al. [2014]: uma *product-based*, uma *family-based* e uma *family-product-based* (Capítulo 4).

- Uma linha de produtos de ferramentas para análise de linhas de produtos (Capítulo 3), a qual implementa as três estratégias formalizadas e duas outras ainda não formalizadas — uma *feature-family-based* e uma *feature-product-based*. Essa linha de produtos, denominada ReAna-SPL, encontra-se publicamente disponível como software livre e de código aberto em `https://github.com/SPLMC/reana-spl`. Até onde sabemos, essa é a primeira ferramenta de *model checking* para linhas de produtos a implementar as três dimensões de análise (*product-based*, *family-based* e *feature-based*).

- Provas de comutatividade entre as estratégias *product-based*, *family-based* e *family-product-based* (Seção 4.2). Isso estabelece sua corretude e aprimora o entendimento vigente sobre o relacionamento entre estratégias para análise de linhas de produtos.

- Um princípio geral para alçar análises de software à operação sobre linhas de produtos usando diagramas de decisão algébricos (Seção 4.2.2.2, Teorema 2).

1.4 Estrutura

Este trabalho está organizado da seguinte forma:

- O Capítulo 2 apresenta conceitos fundamentais à discussão que se segue. Ele introduz linhas de produtos de software e a taxonomia de análise correspondente, assim como os modelos comportamentais paramétricos e diagramas de decisão empregados por nossas técnicas de análise.

- O Capítulo 3 apresenta nossa metodologia de pesquisa, assim como um registro das fases intermediárias do trabalho. Esse capítulo apresenta, ainda, a linha de produtos de ferramentas para análise de linhas de produtos resultante.

- O Capítulo 4 apresenta nossa formalização dos modelos comportamentais para linhas de produtos de software (Seção 4.1) e das nossas estratégias de análise (Seção 4.2). Esse capítulo também enuncia a corretude dessas estratégias como teoremas, assim como apresenta as demonstrações correspondentes.

- No Capítulo 5 são discutidos trabalhos relacionados e futuros, assim como nossas conclusões e ameaças à sua validade.

- O Apêndice A contém a versão completa dos modelos probabilísticos utilizados em exemplos ao longo do texto.
- Por fim, o Apêndice B é uma compilação de grafos de dependências para os principais teoremas apresentados neste trabalho. Tais diagramas foram utilizados ao longo da pesquisa para avaliar o impacto de mudanças, mas também são úteis para visualizar o relacionamento entre os elementos da nossa teoria.

Chapter 2

Background

To better understand the problem and the proposed solution, it is useful to bear in mind concepts regarding software product lines (Section 2.1), particularly software analysis applied to product-line engineering (Section 2.1.4). Within this domain, this work focuses on reliability analysis based on probabilistic behavioral models (Section 2.2).

This chapter lays these conceptual foundations for our research. Furthermore, we provide background on Algebraic Decision Diagrams (Section 2.3), since this type of data structure plays an important role in our analysis techniques.

2.1 Software Product Lines

In the software industry, there are cases in which programs have to be adapted to different platform requirements, such as hardware or operating system. For instance, different versions of an operating system can be created to cope with different processor instruction sets. These *program variants* can be functionally equal, but that is not always the case. No version of our operating system can provide an interface to a graphics card if the host computer does not have one.

At times, the creation of different versions of a software is motivated by variant requirements. As an example, enterprise software can be subject to company-specific business processes or even platforms (e.g., different enterprise databases). In general, this tailoring of software to customer needs, known as *customization*, gives rise to as many coexisting versions of a program as there are customers.

A possible approach to build such program variants is to develop each of them separately. Although this *clone-and-own* approach is sometimes used in practice [Apel et al., 2013a], it is time-consuming and error-prone. For instance, variants realized as separate copies of the source code can have inconsistent evolution of common functionalities, or a bugfix in one variant may not be propagated to the others.

An alternative approach is to view alternative programs that perform the same task, or similar programs that perform similar tasks, as constituents of a *program family* [Dijkstra, 1971]. Regarding similar programs as family members, instead of textual modifications of one another, allows a view that they are modifications of a common ancestor. Such a view has the goal to share code (and corresponding correctness proofs) between programs as far as possible, and to ease their maintenance by isolating the parts that are inherently different.

A realization of the program family view, addressing the issues of the clone-and-own approach, is the *software product line* approach: having a collection of reusable assets from which variants are systematically (or even automatically) generated. The Linux kernel, for instance, is managed according to this approach [Sincero et al., 2007]. Its assets are C headers and source files, whose variability is handled by conditional compilation of certain code regions—using CPP (C Preprocessor) directives. An utility tool is used to select the desired functionality, from which corresponding CPP directives are evaluated and the resulting processed source code is compiled, thereby yielding a custom Linux version. Valid combinations of functionality are described in the Kconfig language, to ensure implementation consistency.

2.1.1 Main Concepts

A Software Product Line is defined as a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [Clements and Northrop, 2001]. Thus, software product line engineering can be seen as the set of processes and techniques used for systematically managing these common features, which provides for improved quality, mass customization capability and reduced costs and time to market [Apel et al., 2013a; Pohl et al., 2005; van der Linden et al., 2007].

The main concern in product-line engineering is managing variability, which is defined by van Gurp et al. [2001] as the ability to change or customize a system. To accomplish this, it is useful to abstract variability in terms of *features*. The concept of a feature encompasses both intentions of stakeholders and implementation-level concerns, and has been subject to a number of definitions [Apel et al., 2013a]. Synthetically, it can be seen as a characteristic or end-user-visible behavior of a software system.

Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, as well as to guide structure, reuse, and variation across all phases of the software life cycle [Apel et al., 2013a]. The features of a product line and their relationships are documented in a *feature model* [Czar-

necki and Eisenecker, 2000; Kang et al., 1990], which can be graphically represented as a *feature diagram*. Throughout this work, we focus on propositional feature models, that is, feature models whose semantics is based on propositional logic. For a feature model FM, we denote its set of features by F. Each feature in this set has a name; feature names are used as atomic propositions to express feature relationships as propositional logic formulae. As an example, one can state $\texttt{f} \Rightarrow \texttt{g}$, meaning that, whenever a product exhibits feature `f`, it must also provide feature `g`.

Figure 2.1 shows an example of propositional feature model, taken from the Body Sensor Network (BSN) product line [Rodrigues et al., 2015]. Each product of this product line is a network of connected sensors that capture vital signs from an individual and send these signs to a central system, which analyzes the data collected and identifies critical health situations. The `Root` feature is, by definition, present in all configurations. Its children are marked as *mandatory*, meaning they must be present whenever its parent is selected. A child feature could also be marked as *optional*, meaning it could be either present or absent in any valid cofiguration.

The domain-related features are grouped under `Monitoring`, which is further broken down into mandatory features `Sensor` and `SensorInformation`. `Sensor` groups features related to the available body sensors. These sensor-related features are *OR-features*, meaning that *at least one* of them must be selected whenever their parent is selected, but multiple selection is also allowed. The same happens for `SensorInformation` and its children, but, since these features correspond to vital signs that result from processing raw sensors data, we must be able to constrain their presence to the presence of the corresponding sensors. These crosscutting concerns are represented by *cross-tree constraints* (below the feature model tree), which are propositional formulae relating features that are not siblings in the diagram.

BSN's feature model also handles persistence of sensor data (`Storage` feature). The supported media are SQLite or in-memory databases, represented by the features `SQLite` and `Memory`, respectively. These features are marked as *alternative*, which means a BSN system must support *exactly one* of them.

A given software system in a product line is referred to as a *product* and is specified by a *configuration*, which is taken as input in the product generation process. A configuration is a selection of features respecting the constraints established by the feature model, and, as such, is represented by a set of atoms: a positive atom denotes feature presence, whereas a negative (or absent) atom denotes feature absence. We denote the set of configurations over a feature set F as C. This set contains all $2^{|F|}$ combinations of feature atoms, each of which must appear in either positive or negative form, but never both. Valid configurations, that is, configurations that satisfy the constraints expressed by the

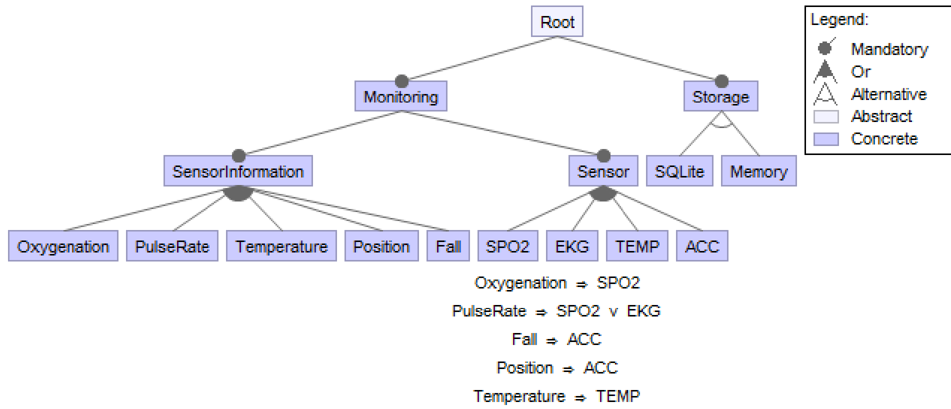


Figure 2.1: Feature model of the BSN product line [Rodrigues et al., 2015].

feature model FM, are denoted by $[\![FM]\!] \subseteq C$. Each $c \in [\![FM]\!]$ specifies the features of a product of the product line.

In the BSN example, let c_1 and c_2 be such that:

$$c_1 = \{\texttt{Root}, \texttt{Monitoring}, \texttt{Sensor}, \texttt{ACC}, \texttt{SensorInformation}, \texttt{Position}, \texttt{Storage}, \texttt{SQLite}\}$$

$$c_2 = \{\texttt{Root}, \texttt{Monitoring}, \texttt{Sensor}, \texttt{EKG}, \texttt{SensorInformation}, \texttt{Position}, \texttt{Storage}, \texttt{SQLite}\}$$

Since both c_1 and c_2 are sets whose elements are in the feature set F, both are configurations ($c_1, c_2 \in C$). However, only c_1 is a *valid* configuration ($c_1 \in [\![FM]\!]$), since c_2 does not satisfy the penultimate cross-tree constraint of the feature model in Figure 2.1 ($c_2 \not\models (\texttt{Position} \Rightarrow \texttt{ACC})$). In other words, there is no use in generating a body sensor network that is able to process accelerometer data to determine the patient's position, and yet is not able to actually *read* the accelerometer.

In a product line, a product comprises a set of assets (e.g., source code files, test cases, documentation), which are derived from a common set known as the *asset base*. The mapping between a given configuration and the assets which compose the corresponding product is called *configuration knowledge* [Czarnecki and Eisenecker, 2000]. Such a configuration knowledge may consist of selecting source files, for instance, but may also handle processing tasks over the selected assets, such as running the C Preprocessor. The locations within the assets where variation occurs are called *variation points*.

Given a configuration, an asset base and a configuration knowledge, the process by which a product is built is called *product derivation* [Apel et al., 2013a]. Actual behavior is included or excluded from a generated product by means of *presence conditions*, which are propositional formulae over features [Czarnecki and Pietroszek, 2006]. For example, when variability is implemented by means of CPP directives, as in the Linux kernel, such presence conditions may be realized using Boolean logic operators over macros that

correspond to features. The derivation process then consists of mapping a configuration to CPP macros, running CPP itself to test `#if` and `#ifdef` directives against the given evaluation of macros, and then compiling the preprocessed source code.

The use of arbitrary (not only atomic) propositions for presence conditions is a means to switch behavior that is conditioned on more than one feature. To operationalize satisfaction of presence conditions, we need to define Boolean functions over feature selections. This way, we also denote a configuration $c \in [\![FM]\!]$ as a Boolean tuple in $\mathbb{B}^{|F|}$, where $\mathbb{B} = \{0, 1\}$ is the set of Boolean values (where 0 and 1 denote the Boolean values `FALSE` and `TRUE`, respectively). Such Boolean tuples have a fixed position for each feature, denoting feature presence or absence by the values 1 and 0 in the respective position. In the upcoming discussion, whenever we refer to k-ary Boolean functions, we assume that Boolean k-tuples can be used as arguments.

2.1.2 Adoption Strategies

To adopt software product line engineering practices, Krueger [2002] identified three possible strategies:

Proactive. Develop a product line from scratch based on careful analysis and design methods. This strategy roughly resembles the waterfall methodology for single-software development [Royce, 1987].

Extractive. Incrementally refactor a collection of existing products to form a product line, extracting the common and varying parts of assets.

Reactive. Extend the product line incrementally on demand.

Those strategies, each of which presents its own trade-offs, can be combined as needed. For instance, a software product line developed by means of a proactive approach eventually meets new requirements, which can be addressed using a reactive approach. Alves et al. [2007] propose a method for product-line adoption which relies on extracting a product line and then incrementally evolving it with a reactive approach.

2.1.3 Variability Implementation

We have seen examples of variability handling by means of CPP directives. Other techniques are also used to implement variability, and those techniques are classified under three dimensions [Apel et al., 2013a]:

Binding time. This dimension refers to the phase during product derivation in which the existing variability is resolved. This can happen before or during compilation

(*compile-time* or *static* variability), at program startup (*load-time* variability) or during execution (*run-time* variability). The ability to perform each of those is closely related to the other dimensions.

Technology. Variability can be realized by means of tools specially built for this purpose (e.g., a preprocessor), but can also rely on programming language constructs (e.g., run-time parameters and *if-then-else* blocks). These approaches are called respectively *tool-based* and *language-based*.

Representation. The means by which variability is expressed in the assets.

Annotation-based (or *annotative*) approaches consist of annotating common assets with tags corresponding to features, such that product derivation can be done by removing the parts annotated with the features which are not selected.

Composition-based (or *compositional*) approaches tackle the variability in a modular way by segregating asset-parts that correspond to each feature in composable units. The ones corresponding to selected features in a given configuration are combined to derive a product.

Other authors also identify a form of variability representation known as *transformation-based* [Haber et al., 2013; Turnes et al., 2011], which relies on transformations over base assets. These transformations are usually performed at the syntactic level, but this is not a formal restriction of this category of techniques.

An usual annotative technique is the use of preprocessor directives, which is the variability representation mechanism in the Linux Kernel [Passos et al., 2013]. This choice of representation naturally limits the possible technology and binding time to a compile-time tool-based approach. Nonetheless, flow-control directives allow a run-time annotation-based and language-based variability implementation.

As for compositional methods, we can see a plug-in framework as an instance of language-based load-time approach. In the realm of tool-based compile-time approaches, there are two main composition mechanisms of interest to product line engineering:

Aspect-Oriented Programming. [Kiczales et al., 1997] This technique aims at the modularization of cross-cutting concerns, i.e., concepts which are necessarily scattered across the implementation of other concerns. These cross-cutting concerns are implemented in modules named *aspects*, which are woven into the main program based on the specification of the points which they affect.

Feature-Oriented Programming. [Batory et al., 2004; Prehofer, 1997] This is a technique by which the concepts in a program are implemented in modules, each of

which is associated to a feature. Product derivation is thus carried out by incrementally composing these so called *feature modules* into the result of the previous composition, yielding at each step a program which increments the previous one with the refinements in the given feature. A feature module can add new classes and members, as well as override existing methods.

Delta-Oriented Programming [Schaefer et al., 2010] is a well-know example of transformation-based (or *transformational*) approach [Haber et al., 2011]. It is similar to Feature-Oriented Programming, but the modules (*deltas*) are also capable of removing classes and members. Additionally, the deltas are not mapped one-to-one into features. Instead, there is an explicit language construct for specifying dependencies between them and predicates over the selected features which must hold true for a given delta to be applicable.

So far, we presented examples of source-code variability handling. However, these implementation techniques can also be used to handle different kinds of assets. For instance, a compositional approach, similar to aspect-oriented programming, was used to handle variability in use cases [Almeida and Borba, 2009] and business processes [Turnes et al., 2011]. Teixeira et al. [2015] also exploited compositional variability handling, in the context of a product line of theories described using the specification language of the Prototype Verification System (PVS) [Owre et al., 2001].

2.1.4 Product-Line Analysis

Analysis of software product lines is a broad subject, in the sense that it can refer to verification of any of the product line artifacts, including the feature model and the configuration knowledge [Apel et al., 2013a]. Hence, we focus on verification of the possibly derivable products. This does not necessarily mean generating all products in a product line and analyzing each of them, as long as analyzed properties can be somehow generalized to the product line as a whole. We refer to the latter case as *variability-aware analysis*.

2.1.4.1 Techniques

As with single-system analysis, product line analyses can be performed statically (at compilation time or before) or at execution time. Although run-time analyses such as unit and integration testing have been applied in the context of software product lines [Silveira Neto et al., 2011], we examine only analyses which apply statically. This is a design decision for our research, based on the availability of static analysis tools and on ongoing activity within our research group regarding this kind of technique.

Thüm et al. [2014] performed a survey on static analyses of software product lines in which four main classes where identified:

Type checking. Analysis of well-typedness of a program with respect to a given *type system* [Pierce, 2002]. It captures errors such as mismatched method signatures and undeclared types, which are prone to happen if features can add or remove methods and classes.

Model checking. Consists of systematically exploring the possible states in a formal model of the system, to find out whether it satisfies a given property [Baier and Katoen, 2008]. Some model checkers operate directly on source code, while others allow other abstractions of the system's behavior (e.g., Markov chains).

Static analysis. Based on compile-time approximation of the run-time behavior of a program, such as in data-flow and control-flow analyses. This type of analysis usually involves the verification of source code and can signal problems such as access to uninitialized memory regions.

Theorem proving. Relies on encoding system properties as theories and specifications of its desired behavior as theorems. These theorems then need to be proved in order to assert the modeled system is correct, i.e., it satisfies the specified properties. The theories and theorems may be specified using the language of a proof assistant such as PVS [Owre et al., 2001], or can be generated from invariant specifications declared in the source code using JML [Leavens and Cheon, 2006], for instance.

2.1.4.2 Strategies

Thüm et al. [2014] define three analysis strategies for product lines, i.e., approaches for applying the aforementioned analysis techniques to a software product line as a whole. Those strategies are the following:

Product-based. Consists of analyzing derived products or models thereof. This can be accomplished by generating all such products (the *brute-force* approach) or by sampling them based on some coverage criteria (e.g., covering pair-wise or triple-wise feature interaction). The main advantage of this strategy is that the analysis can be performed exactly as in the single-system case by off-the-shelf tools. However, the time and processing cost can be prohibitively large (exponential blowup) if the considered product line has a great number of products.

Feature-based. Analyzes all domain artifacts implementing a given feature in isolation, not considering how they relate to other features. However, issues related to feature interactions are frequent, which renders false the premise that features can be

modularly analyzed. In spite of this, this approach is able to verify compositional properties (e.g., syntactic correctness) and has the advantage of supporting *open-world scenarios* — since a feature is analyzed in isolation, not all features must be known in advance.

Family-based. Operates only in domain artifacts, usually merging all variability into a single *product simulator* (also known as *virtual product* or *metaproduct*). This simulator is then analyzed by considering only valid combinations of the features as specified in the feature model. It is possible, for instance, to compose feature modules by encoding their variability as *if-then-else* blocks and dispatcher methods and then apply off-the-shelf software model checking [Apel et al., 2013b].

There is also the possibility to employ more than one strategy simultaneously. In this way, weaknesses resulting from one approach can be overcome by the application of another. This is particularly useful for feature-based approaches, which are generally not sufficient due to feature interactions.

For instance, Thüm et al. [2011] proposes formal verification of design-by-contract properties [Meyer, 1992] restricted to feature modules. This would be characterized as a feature-based strategy, but after product derivation the proof obligations that are verified feature-wise can be changed due to source code transformation. Hence, each product is derived to generate the complete proof obligations. Nonetheless, most of the proofs obtained in the feature-based phase can be reused, so this composite strategy can be seen as *feature-product-based*.

Similarly, it is possible to derive *feature-family-based*, *family-product-based* and even *feature-family-product-based* strategies, although the aforementioned survey did not find any case of the latter in the literature.

2.2 Reliability Analysis

We define the reliability of a system as the probability that, starting from an initial state, the system eventually reaches a set of *target* (also *success*) states. This reliability value is called *reachability probability*. The property that a system presents a reachability probability within given bounds is defined as a probabilistic existence property [Grunske, 2008]. This class of properties is specified using Probabilistic Computation Tree Logic (PCTL) [Hansson and Jonsson, 1994] as $P_{\bowtie p}[\Diamond\, \Phi]$, where p is a probability, $\bowtie \in \{=, <, \leq, >, \geq\}$, Φ is a propositional formula that can be evaluated for a system state, and $\Diamond$ is the temporal logic "eventually" operator.

Our particular goal is to compute the reliability of a system, instead of checking that it lies within certain limits. To perform this computation, we first model the system's behavior as a Discrete-time Markov Chain (DTMC)—a tuple $(S, s_0, \mathbf{P}, T)$, where S is a set of states, $s_0 \in S$ is the initial state, $\mathbf{P}$ is the transition probability matrix $\mathbf{P} : S \times S \to [0, 1]$, and $T \subseteq S$ is the set of target states[1]. Each row of the transition probability matrix sums to 1, meaning that, for every state $s \in S$, the probabilities of transitioning from s to all states $t \in S$ (including s itself) must sum to 1. Reliability analysis is then the process through which we determine the probability p for which it holds that $P_{=p}[\lozenge success]$, where *success* is a proposition that only holds true for $s \in T$.

A DTMC can be seen as a graph in which nodes represent states and edges represent transitions. Every non-zero entry (s, s') in the transition probability matrix $\mathbf{P}$ is represented by a labeled transition $s \xrightarrow{p} s'$ in this graph, where $p = \mathbf{P}(s, s')$. Figure 2.2 presents an example of DTMC viewed as a graph. In this view, the reachability probability is the sum of the probabilities along every possible path from the initial state (blue node) to the success state (green node). The equation on the left-hand side of this figure depicts this summation, with each term corresponding to one of the three possible paths (note the correspondence between the red highlighted term and the red highlighted path, for instance).

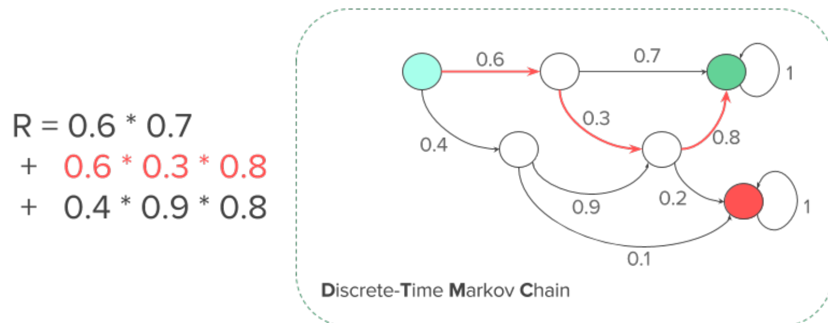


Figure 2.2: Example graph view of a DTMC and the corresponding reachability probability.

The reachability probability for a DTMC can be computed using probabilistic model checking algorithms, implemented by off-the-shelf tools such as PRISM [Kwiatkowska et al., 2011] and PARAM [Hahn et al., 2010]. An intuitive and correct view of reachability probability, although not well-suited for efficient implementation, is that a target state is reached either directly or by first transitioning to a state that is able to recursively reach it. We present a formalization of this property, adapted from Baier and Katoen [2008], that suits the purpose of this work.

[1] This definition departs from the one by Baier and Katoen [2008] in that it abstracts away the possibility of multiple initial states and the computation of other temporal properties, while incorporating the target states in the model. This view suits our goal to focus on reliability analysis.

Property 1 (Reachability probability for DTMCs)**.** Given a DTMC $\mathcal{D} = (S, s_0, \mathbf{P}, T)$, a state $s \in S$, and a set T of target states that are reachable from s ($s \notin T$), the probability of reaching a state $t \in T$ from s, denoted by $Pr^{\mathcal{D}}(s, T)$, satisfies the following property:

$$Pr^{\mathcal{D}}(s, T) = \sum_{s' \in S \setminus T} \mathbf{P}(s, s') \cdot Pr^{\mathcal{D}}(s', T) + \sum_{t \in T} \mathbf{P}(s, t)$$

If $s \in T$, then $Pr^{\mathcal{D}}(s, T) = 1$. If T is not reachable from s, then $Pr^{\mathcal{D}}(s, T) = 0$. For brevity, whenever T is a singleton $\{t\}$, we write $Pr^{\mathcal{D}}(s, t)$ to denote $Pr^{\mathcal{D}}(s, T)$.

In a product line, different products give rise to distinct behavior models. To handle the behavior variability that is inherent to product lines, we resort to *Parametric Markov Chains* [Daws, 2005].

2.2.1 Parametric Markov Chains

Parametric Markov Chains (PMC) extend DTMCs with the ability to represent variable transition probabilities. Whereas probabilistic choices are fixed at modeling time and represent possible behavior that is unknown until run time, variable transitions represent behavior that is unknown already at modeling time. These variable transition probabilities can be leveraged to represent product-line variability [Chrszon et al., 2016; Ghezzi and Molzam Sharifloo, 2013; Rodrigues et al., 2015].

Definition 1 (Parametric Markov Chain)**.** A Parametric Markov Chain is defined by Hahn et al. [2011] as a tuple $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$, where S is a set of states, s_0 is the initial state, $X = \{x_1, \ldots, x_n\}$ is a finite set of parameters, $\mathbf{P}$ is the transition probability matrix $\mathbf{P} : S \times S \to \mathcal{F}_X$, and $T \subseteq S$ is the set of *target* (or *success*) states. The set $\mathcal{F}_X$ comprises the *rational expressions* over $\mathbb{R}$ with variables in X, that is, fractions of polynomials with Real coefficients. This way, the semantics of a rational expression ε is a *rational function* $f_\varepsilon(x_1, \ldots, x_n) = \frac{p_1(x_1, \ldots, x_n)}{p_2(x_1, \ldots, x_n)}$ from $\mathbb{R}^n$ to $\mathbb{R}$, where p_1 and p_2 are Real polynomials. For brevity, we hereafter refer to rational expressions simply as *expressions*.

By attributing values to the variables, it is possible to obtain an ordinary (non-parametric) DTMC. Parameters are given values by means of an *evaluation*, which is a total function[2] $u : X \to \mathbb{R}$ for a set X of variables. For an expression $\varepsilon \in \mathcal{F}_X$ and an evaluation $u : X' \to \mathbb{R}$ (where X' is a set of variables), we define $\varepsilon[X/u]$ to denote the expression obtained by replacing every occurrence of $x \in X \cap X'$ in ε by $u(x)$, also denoted by $\varepsilon[x_1/u(x_1), \ldots, x_n/u(x_n)]$. Note that, if u's domain, X', is different from the set X of variables in ε, then $\varepsilon[X/u] = \varepsilon[(X \cap X')/u]$.

Definition 2 (Expression evaluation)**.** Given expressions ε_1 and ε_2 over variables sets X_1 and X_2, respectively, let $X \supseteq X_1 \cup X_2$ be a set of variables, $x \in X$ be a variable, $c \in \mathbb{R}$ and $n \in \mathbb{N}$ be constant values, and $u : X \to \mathbb{R}$ be an evaluation. Expression evaluation is defined inductively as follows:

$$\frac{\varepsilon_1}{\varepsilon_2}[X/u] = \frac{\varepsilon_1[X/u]}{\varepsilon_2[X/u]} \qquad (\varepsilon_1 \times \varepsilon_2)[X/u] = \varepsilon_1[X/u] \times \varepsilon_2[X/u]$$

$$(\varepsilon_1 + \varepsilon_2)[X/u] = \varepsilon_1[X/u] + \varepsilon_2[X/u] \qquad (\varepsilon_1 - \varepsilon_2)[X/u] = \varepsilon_1[X/u] - \varepsilon_2[X/u]$$

$$x[X/u] = u(x) \qquad \varepsilon_1^n[X/u] = \varepsilon_1[X/u]^n$$

$$c[X/u] = c$$

This definition can be extended to substitutions by other expressions. Given two variable sets X and X', their respective induced sets of expressions $\mathcal{F}_X$ and $\mathcal{F}_{X'}$, and an expression $\varepsilon \in \mathcal{F}_X$, a generalized evaluation function $u : X \to \mathcal{F}_{X'}$ substitutes each variable in X for an expression in $\mathcal{F}_{X'}$. The generalized evaluation $\varepsilon[X/u]$ then yields an expression $\varepsilon' \in \mathcal{F}_{X'}$. Moreover, successive expression evaluations can be thought of as rational function compositions: for $u : X \to \mathcal{F}_{X'}$ and $u' : X' \to \mathbb{R}$,

$$\varepsilon[X/u][X'/u'] = \varepsilon[x_1/u(x_1)[X'/u'], \ldots, x_k/u(x_k)[X'/u']] \tag{2.1}$$

for $x_1, \ldots, x_k \in X$ (since u is a total function, we do not need to consider non-evaluated variables).

The PMC induced by an evaluation u is denoted by $\mathcal{P}_u = (S, s_0, \emptyset, \mathbf{P}_u, T)$ (alternatively, $\mathcal{P}[X/u]$), where $\mathbf{P}_u(s, s') = \mathbf{P}(s, s')[X/u]$ for all $s, s' \in S$. To ensure the resulting chain after evaluation is indeed a valid DTMC, one must use a *well-defined* evaluation.

Definition 3 (Well-defined evaluation)**.** An evaluation $u : X \to \mathbb{R}$ is *well-defined* for a PMC $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$ iff, for all $s, s' \in S$, it holds that

- $\mathbf{P}_u(s, s') \in [0, 1]$ (all transitions evaluate to valid probabilities)
- $\mathbf{P}_u(s, \mathit{Succ}(s)) = 1$ (stochastic property—the probability of disjoint events must add up to 1)

In this definition, $\mathit{Succ}(s) = \{s' \in S \mid \mathbf{P}_u(s, s') \neq 0\}$ is the set of successor states of s, and $\mathbf{P}(s, S) = \sum_{s' \in S} \mathbf{P}(s, s')$.

Hereafter, we drop explicit mentions to well-definedness whenever we consider an evaluation or a DTMC induced by one, because we are only interested in this class of

[2] Hahn et al. [2011] actually define it in a more general way as a partial function. However, for our purpose, it suffices to consider total functions.

evaluations. Nonetheless, we still need to prove that specific evaluations are indeed well-defined.

2.2.2 Parametric Probabilistic Reachability

To compute the reachability probability in a model with variable transitions, we use a parametric probabilistic reachability algorithm. A parametric model checking algorithm for probabilistic reachability takes a PMC $\mathcal{P}$ as input and outputs a corresponding expression ε representing the probability of reaching its set T of target states. Figure 2.3 presents the intuition of computing such an expression, following the same mapping from terms to paths that we used for DTMCs (Figure 2.2).

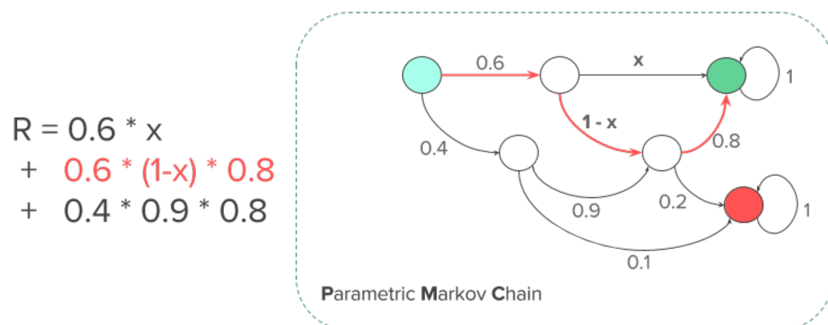


Figure 2.3: Example graph view of a PMC and the intuition for the corresponding reachability probability expression.

Hahn et al. [2011] present a parametric probabilistic reachability algorithm (Algorithm 1) and prove that evaluating the resulting expression ε with an evaluation u yields the reachability probability for the DTMC induced in $\mathcal{P}$ by the same evaluation u. The main idea is that, for a given state s, the probability of one of its predecessors ($s_{pre} \in Pre(s)$) reaching one of its successors ($s_{succ} \in Succ(s)$) is given by the sum of the probability of transitioning through s and the probability of bypassing it.

Algorithm 1 Parametric Reachability Probability for PMCs [Hahn et al., 2011]

Require: PMC $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$. States $s \in T$ are absorbing. For all $s \in S$, it holds that s is reachable from s_0 and T is reachable from s.

1: **for all** $s \in S \setminus (\{s_0\} \cup T)$ **do**
2:     **for all** $(s_{pre}, s_{succ}) \in Pre(s) \times Succ(s)$ **do**
3:         $\mathbf{P}(s_{pre}, s_{succ}) = \mathbf{P}(s_{pre}, s_{succ}) + \mathbf{P}(s_{pre}, s) \cdot \frac{1}{1-\mathbf{P}(s,s)} \cdot \mathbf{P}(s, s_{succ})$
4:     **end for**
5:     $eliminate(s)$
6: **end for**
7: **return** $\frac{1}{1-\mathbf{P}(s_0,s_0)}\mathbf{P}(s_0, T)$

For such a pair of predecessor and successor states, we update the transition probability matrix with the newly computed value (Line 3):

$$\underbrace{\mathbf{P}(s_{pre}, s_{succ})}_{\text{update}} = \overbrace{\mathbf{P}(s_{pre}, s_{succ})}^{\text{bypass}} + \overbrace{\underbrace{\mathbf{P}(s_{pre}, s)}_{\text{reach } s} \cdot \frac{1}{1 - \underbrace{\mathbf{P}(s, s)}_{\text{stay at } s}} \cdot \underbrace{\mathbf{P}(s, s_{succ})}_{\text{leave } s}}^{\text{go through } s}$$

Once this computation has been performed for all predecessor ($Pre(s)$) and successor states ($Succ(s)$), s itself is *eliminated* from the set S of states, and the process starts again by arbitrarily picking another state. Figure 2.4 [Hahn et al., 2011] illustrates the update of the transition probability matrix for a given state s and a single pair of predecessor and successor states. In this example, other states and respective transitions are omitted. Note that, since there is a self-loop with probability p_c, there are infinite possible paths going through s, each corresponding to a number of times the loop transition is taken before transitioning to s_{succ}. Hence, the sum of probabilities for these paths correspond to the infinite sum $\sum_{i \in \mathbb{N}} p_a (p_c)^i p_b = p_a (\sum_{i \in \mathbb{N}} p_c^i) p_b = p_a \frac{1}{1-p_c} p_b$.

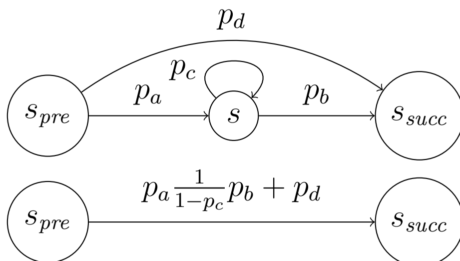


Figure 2.4: Elimination of state s in the parametric reachability probability algorithm (adapted from Hahn et al. [2011]).

Lemma 1 (Parametric probabilistic reachability soundness)**.** *Let* $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$ *be a PMC,* u *be a well-defined evaluation for* $\mathcal{P}$*, and* ε *be the output of the parametric probabilistic reachability algorithm by Hahn et al. [2011] (Algorithm 1) for* $\mathcal{P}$ *and* T*. Then,* $Pr^{\mathcal{P}_u}(s_0, T) = \varepsilon[X/u]$*.*

Proof. The algorithm by Hahn et al. [2011] is based on eliminating states until only the initial and the target ones remain. Its proof consists of showing that each elimination step preserves the reachability probability. We refer the reader to the work by Hahn et al. [2011] for more details on the algorithm itself and the proof mechanics. □

2.3 Algebraic Decision Diagrams

An Algebraic Decision Diagram (ADD) [Bahar et al., 1997] is a data structure that encodes k-ary Boolean functions $\mathbb{B}^k \to \mathbb{R}$. As an example, Figure 2.5 depicts an ADD representing the following binary function f:

$$f(x, y) = \begin{cases} 0.9 & \text{if } x \land y \\ 0.8 & \text{if } x \land \neg y \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

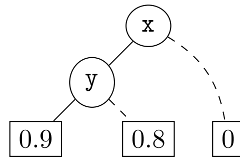


Figure 2.5: ADD A_f representing the Boolean function f in Equation (2.2).

Each internal node in the ADD (one of the circular nodes) marks a decision over a single parameter. Function application is achieved by walking the ADD along a path that denotes this decision over the values of actual parameters: if the parameter represented by the node at hand is 1 (*true*), we take the solid edge; otherwise, if the actual parameter is 0 (*false*), we take the dashed edge. The evaluation ends when we reach a terminal node (one of the square nodes at the bottom).

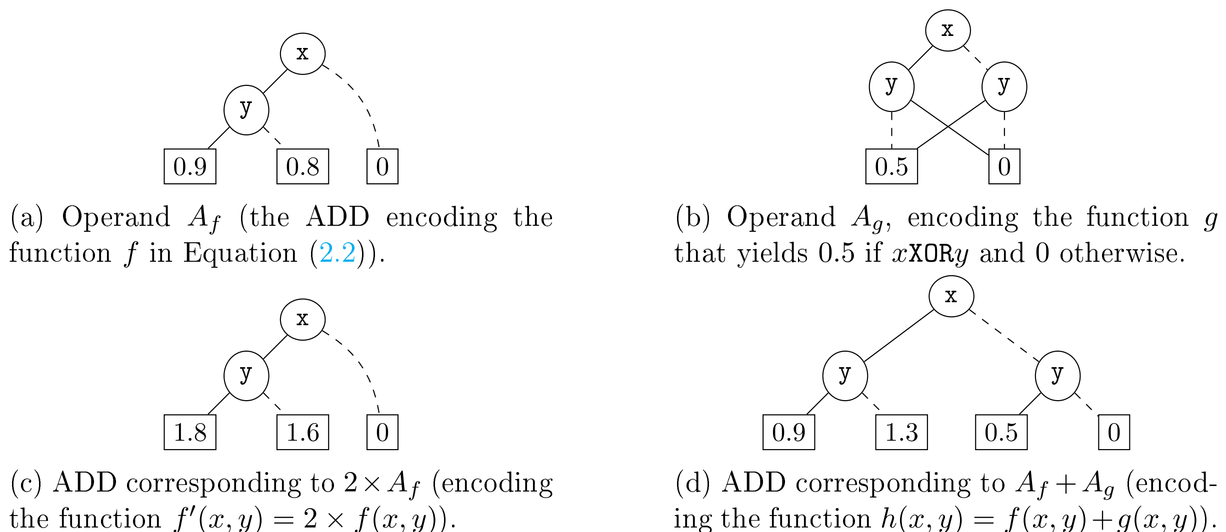
In the example, to evaluate $f(1, 0)$, we start in the `x` node, take the solid edge to node `y` (since the actual parameter x is 1), then take the dashed edge to the terminal 0.8. Thus, $f(1, 0) = 0.8$. Henceforth, we will use a function application notation for ADDs, meaning that, if A is an ADD that encodes function f, then $A(b_1, \ldots, b_k)$ denotes $f(b_1, \ldots, b_k)$. For brevity, we also denote indexed parameters $b_1, \ldots, b_k$ as $\bar{b}$, and the application $A(\bar{b})$ by $[\![A]\!]_{\bar{b}}$.

In our setting, we use ADDs to denote mappings from configurations to Real values. That is, Boolean parameters denote presence or absence of features, and the image of a given configuration is the corresponding reliability value. ADDs have several applications, among which two are of direct interest to this work: arithmetics over Boolean functions and encoding of if-then-else operations over presence conditions.

The first ADD operation of interest relates to efficient application of arithmetics over Boolean functions. The intuition of is that an arithmetic operation over ADDs is equivalent to performing the same operation on corresponding terminals of the operands. Thus, we denote ADD arithmetics by corresponding real arithmetics operators.

In Figure 2.6, we see two examples of ADD arithmetics. The first and simpler one (Figure 2.6c) shows the multiplication of the ADD A_f (Figure 2.6a) by the constant factor 2. This operation takes place by multiplying terminals by the given factor. The second example (Figure 2.6d) shows the sum of ADDs A_f and A_g (Figure 2.6b), yielding an ADD $A_h = A_f + A_g$ such that $A_h(x, y) = A_f(x, y) + A_g(x, y)$. Such an operation is more involved, and its details fall outside the scope of our work.

An important aspect that motivated the use of ADDs for variability-aware arithmetics is that ADD arithmetic operations are linear in the input size. For instance, let us examine an arbitrary arithmetic operation $\odot$ of ADDs A_f and A_g, both on k parameters. Enumerating all valid inputs to the operand functions would take exponential time ($O(2^k)$), whereas ADD arithmetics can be performed in $O(|A_f| \cdot |A_g|)$ (where $|A|$ denotes the *size* of ADD A, that is, its number of internal nodes).



(a) Operand A_f (the ADD encoding the function f in Equation (2.2)).

(b) Operand A_g, encoding the function g that yields 0.5 if xXORy and 0 otherwise.

(c) ADD corresponding to $2 \times A_f$ (encoding the function $f'(x, y) = 2 \times f(x, y)$).

(d) ADD corresponding to $A_f + A_g$ (encoding the function $h(x, y) = f(x, y) + g(x, y)$).

Figure 2.6: Example of an arithmetic operation over ADDs.

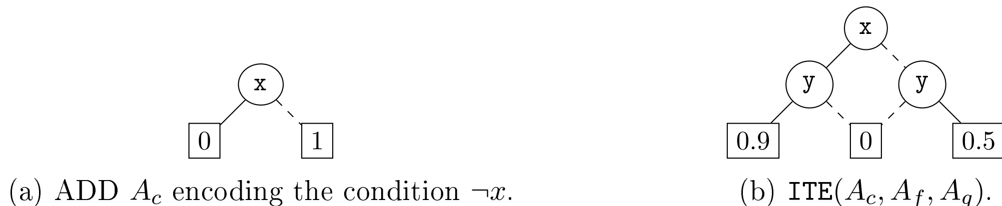
Formally, given a valuation for Boolean parameters $\bar{b} = b_1, \ldots, b_k \in \mathbb{B}^k$, it holds that:

1. $\forall_{\odot \in \{+,-,\times,\div\}} \cdot (A_1 \odot A_2)(\bar{b}) = A_1(\bar{b}) \odot A_2(\bar{b})$
2. $\forall_{i \in \mathbb{N}} \cdot A_1^i(\bar{b}) = A_1(\bar{b})^i$

The second operation of interest is the algorithmic encoding of the result of an *if-then-else* operation over ADDs again as another ADD. For the ADDs A_{cond}, A_{true}, and A_{false}, we define the ternary operator `ITE` (*if-then-else*) as

$$\texttt{ITE}(A_{cond}, A_{true}, A_{false})(c) = \begin{cases} A_{true}(c) & \text{if } A_{cond}(c) \neq 0 \\ A_{false}(c) & \text{if } A_{cond}(c) = 0 \end{cases}$$

This operation, whose time complexity is $O(|A_{cond}| \cdot |A_{true}| \cdot |A_{false})$, is illustrated by Figure 2.7. This figure depicts an ADD resulting from $\texttt{ITE}(A_c, A_f, A_g)$ (Figure 2.7b), where A_c (Figure 2.7a) encodes the function $c(x, y) = \neg x$, and the ADDs A_f and A_g are taken from Figures 2.6a and 2.6b. As with ADD arithmetics, the details of the ADD `ITE` operation are omitted for being out of scope.



(a) ADD A_c encoding the condition $\neg x$.

(b) $\texttt{ITE}(A_c, A_f, A_g)$.

Figure 2.7: Example of an `ITE` operation over ADDs.

Note that we presented the time complexities for the ADD operations in terms of the size of each operand. However, this number is itself dependent upon the ordering of variables, that is, the level of the corresponding decision nodes in the binary tree. Different orderings may need a different number of internal nodes, as depicted by the ADD in Figure 2.8. This ADD encodes the same function f (Equation (2.2)) as the ADD A_f in Figure 2.5, but in this case we have chosen a different ordering of variables—`y` as the root and `x` in the second level. With the chosen ordering, the resulting ADD ended up with 3 internal nodes, as opposed to 2 nodes in the original case.

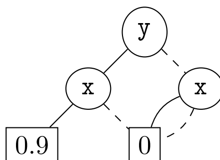


Figure 2.8: Alternative ordering for encoding the Boolean function f in Equation (2.2) as an ADD.

The absolute difference between these alternative orderings was negligible, because the function at hand is only binary. In general, however, given the number k of parameters of the encoded function, the size of an ADD may be $O(k)$ with the best-case ordering, but may also be $O(2^k)$ with the worst-case ordering. Note, however, that not all Boolean functions are subject to exponential orderings, and the same applies to linear orderings. For instance, any ordering of variables of the ADD A_g in Figure 2.6b yields an ADD with 3 internal nodes. More details on this matter and information on ADDs in general can be found in the work of Bahar et al. [1997].

Chapter 3

A Product Line of Product-line Analysis Tools

In this chapter, we present our research method (Section 3.1) and a corresponding record of the intermediate phases of our work. The latter encompasses the implementation of a product line of product-line analysis tools (Sections 3.2 and 3.3) and the analysis of each variant in search for common building blocks (Section 3.4). These implementation and analysis steps led to the discovery of a commuting diagram in the domain of product-line reliability analysis, which we present with further detail in Sections 4.1 and 4.2.

3.1 Research Method

Existing approaches to lifting related analysis techniques to product lines often focus on the family-based dimension [Chrszon et al., 2016; Dubslaff et al., 2015; Midtgaard et al., 2015; von Rhein et al., 2016], relating it only to the product-based dimension to ensure soundness. In the context of reliability analysis, particularly, there is no theory comprising both (a) a formal specification of the three dimensions and resulting analysis strategies, and (b) proof that such analyses commute. To address this issue, we formulated the following research question:

Research Question
Is it possible to obtain equivalent results using different analysis strategies?

To relate the different analysis strategies in the currently accepted taxonomy [Thüm et al., 2014], we decided to take an existing product-line analysis tool as a starting point and then evolve it to a product line of product-line analysis tools. The process of building such a product line would involve the mapping and implementation of variability in the

product-line analysis domain. The resulting product line would then be extended by theoretical assets, in the sense that formal definitions and soundness proofs related to the implemented techniques would be made available for systematic reuse.

The tool of choice was a feature-family-based reliability analyzer, named ReAna, that was under development by our research group. The assumption was that hands-on experience with developing an analysis tool would provide insight into possible variation points for the later development of the product line. Moreover, as the tools' developers all belonged to the research group, communication would be facilitated in this arrangement.

Our research was analytical [Basili, 1993], grounded on secondary studies [Thüm et al., 2014; von Rhein et al., 2013] and on the practical experience gained from the development of the product line of analysis tools. Following guidelines by Sjøberg et al. [2008], an abductive process was used to generate pertinent constructs and relationships based on the practical aspects. Then, deductive processes were applied to achieve formal consistency, that is, the resulting theory comprises unambiguous definitions and sound relationships between them (lemmas and theorems).

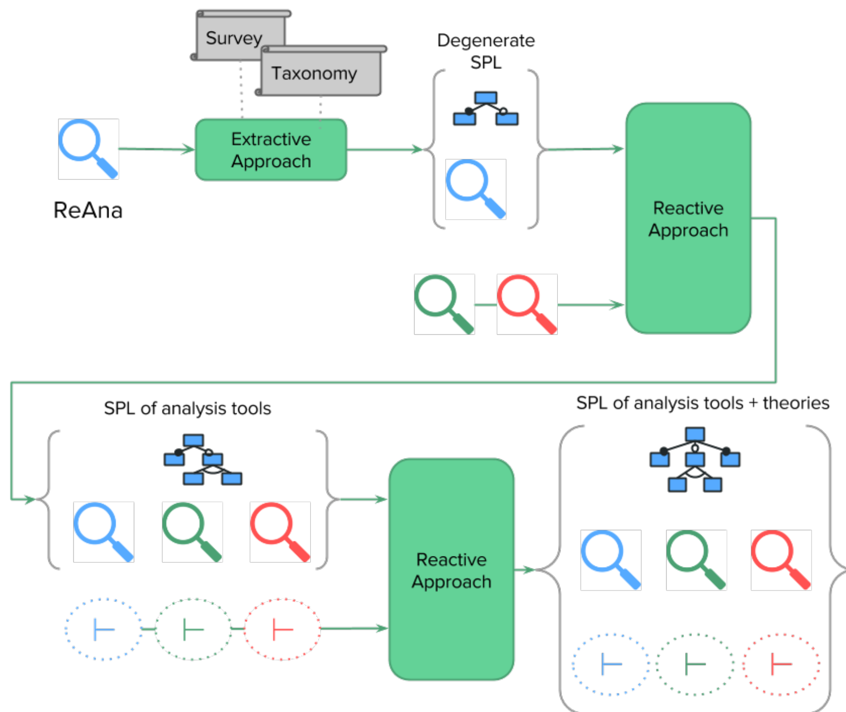


Figure 3.1: Outline of the research phases. Magnifying glasses denote analysis tools, and turnstiles (⊢) denote theories.

Figure 3.1 depicts the research phases, which are summarized by the following steps:

1. Perform a literature review on the domain of software product line analyses, mainly focused on the works considered in the survey performed by Thüm et al. [2014].

2. Contribute to the ongoing implementation of a reliability analysis tool for product lines (ReAna), to gain insight on the mechanisms involved.
3. Define scope limits for considered analyses, regarding their types (e.g., model checking), properties analyzed (e.g., reliability) and program life-cycle phase (e.g., run-time, compile-time).
4. Perform a domain analysis of SPL analysis techniques, identifying variation points in the aforementioned tool. This step should produce a *degenerate* software product line composed of a single analysis tool.
5. Use a reactive approach to extend the single-product product line of analysis tools by adding support to verification of different quality properties and analysis strategies.
6. Empirically investigate the commutativity of the implemented strategies.
7. Prove that the implemented techniques are sound.
8. Analyze the soundness demonstration of the implemented techniques in order to identify common steps, patterns, and underlying principles.

3.1.1 Threats to Validity

The main contribution of this work is analytical, obtained in a deductive way. As such, the validity of the conclusions is conditioned on the validity of the premises and on the correct application of deduction principles. The former concerns whether the formal constructs correspond to the practical ones (*"do the implementation and the theory correspond to one another?"*). The latter concerns the consistency of specifications and correctness of proofs.

To address the validity of the mapping between software constructs and formal definitions, we planned to implement the product line of analysis tools using functional programming principles (although not necessarily using a purely functional programming language). The assumption is that, by organizing the source code into small, manageable modules, with limited presence of side-effects, it is easier to reason about the correctness of definitions and specifications [Backus and John, 1978]. This programming discipline does not *guarantee* a correct mapping between software and mathematical assets, but mitigates the risk of mismatching.

To further reduce the possibility of human mistake, we planned to submit the code, specifications, and proofs for review by fellow researchers. We also planned a submission to a scientific journal whose editorial board members are experienced in the use of formal methods, preferably in the context of software product lines.

Moreover, analytic research should, whenever possible, compare its results with empirical observation [Basili, 1993]. Since our research aims at relating different analysis strategies, we planned to provide empirical evidence that these strategies are indeed equivalent. This assessment consisted of analyzing six product lines using each of the implemented strategies and comparing the numerical results.

Last, we must discuss to what extent our results can be generalized. By construction, we limited our scope to reliability analysis using model checking. Thus, we do not claim our results can be immediately generalized to other types of analysis. On the contrary, we suggest that specific research be conducted towards generalizing the results of this work. In particular, we believe our research method can be used to evolve our product line of analysis tools to support other analysis types, which could provide useful information to the generalization task.

3.2 Domain Engineering

Research steps 2 and 3 were performed simultaneously. While implementing the ReAna tool, we identified a variation point for choosing between the original feature-family-based approach and an alternative feature-product-based approach. With this insight, we delimited the scope to reliability analysis of probabilistic behavioral models (probabilistic model checking). Within this analysis domain, we did not limit analysis strategies—the goal was to explore all strategies in the taxonomy.

Hence, the propositional phase of this work consisted of developing a product line of product-line reliability analysis tools. The products of this product line are instances of the analysis strategies present in the taxonomy by Thüm et al. [2014]. We now report on the construction of this product line using a sequence of extractive and reactive approaches [Krueger, 2002].

3.2.1 Overview

The ReAna tool was designed to perform reliability analysis of product lines, based on probabilistic models of their behavior. These models are parametric Markov chains (PMC), denoting probabilistic changes of state for the execution of a number of products (i.e., the parameters are used to encode variability, as shown in Section 2.2). ReAna takes UML activity and sequence diagrams as input, so that engineers do not have to model the behavior of a product line directly as a PMC.

To enable reliability analysis, the UML behavioral models for a product line have to be annotated with the reliabilities of components. These reliability values are denoted by probabilities in the diagram's messages, using `prob` tags from the UML MARTE

profile [Object Management Group, 2011]. Figure 3.2 is an example of such reliability-annotated diagrams. Each message is annotated (via `prob` tags) with the probability that it will be correctly sent and received. UML's built-in mechanism of `opt` fragments, which usually represents run-time variability, is reframed to express product-line variability (i.e., configurability). Each guard in an `opt` fragment denotes its corresponding presence condition. The enclosing sequence diagram, for instance, is conditioned on the selection of the *oxygenation* feature, whereas the two innermost fragments are conditioned on features *memory* and *sqlite*, respectively.

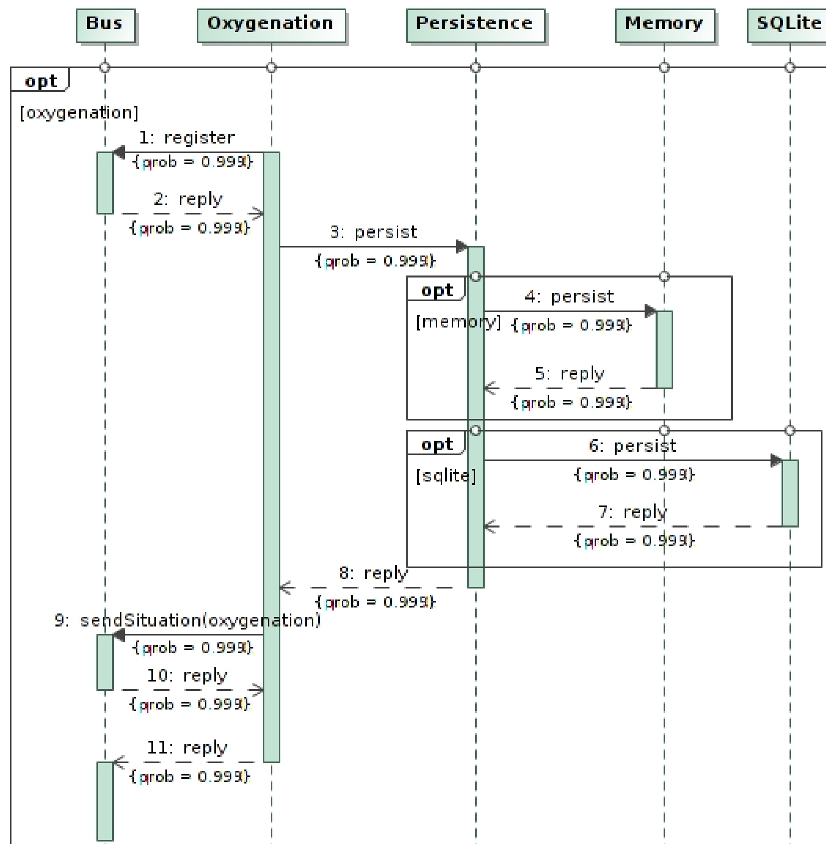


Figure 3.2: Example of reliability-annotated sequence diagram used in a product line. The `prob` tags annotate each message with its reliability, and `opt` fragments mark variation points with corresponding presence conditions as guards.

ReAna transforms the input UML behavioral models into PMCs according to rules defined elsewhere [Ghezzi and Molzam Sharifloo, 2013; Rodrigues et al., 2015]. During this transformation, the behavior described by each fragment is represented by a PMC, in which the behavior of each nested fragment is abstracted using a unique variable. We say each abstraction by a variable is a dependency on the corresponding PMC. Hence, the transformation ends with a set of PMCs bound together by a dependency relation. The presence condition associated with each fragment (on the guard condition) is *separately*

associated to the corresponding PMC (i.e., variables do *not* encode presence conditions, but rather serve as identifiers).

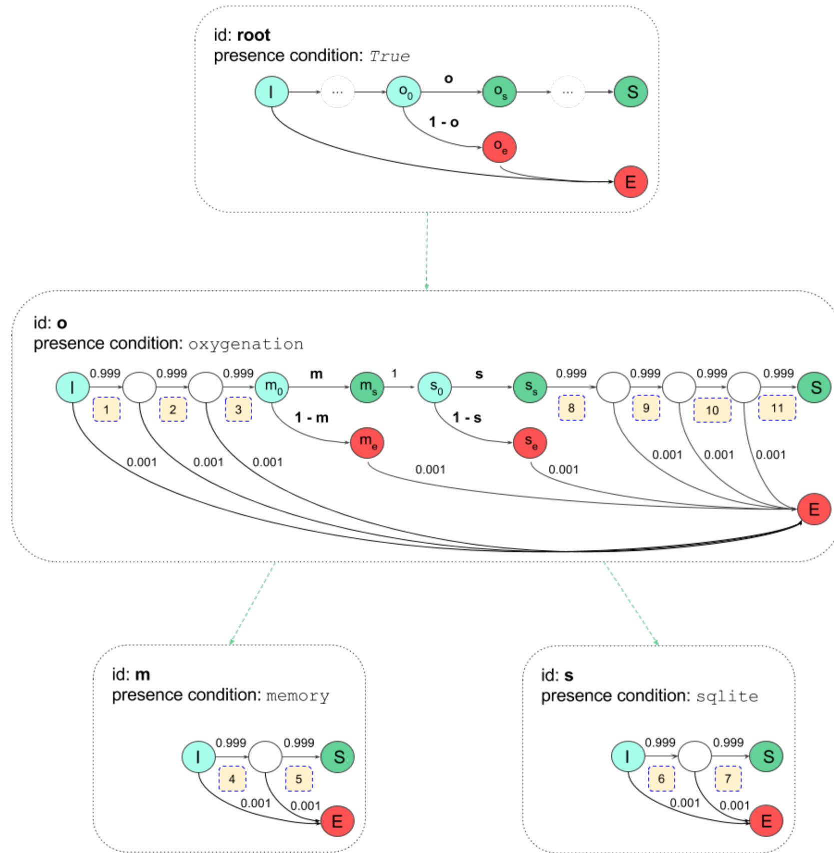


Figure 3.3: Example of PMCs obtained from the sequence diagram in Figure 3.2. Dashed green arrows denote the dependency relation.

Figure 3.3 depicts the PMCs obtained from the UML models in Figure 3.2 using ReAna's transformation process. In each PMC, the blue state labeled with "I" is the initial state, the success state is green with label "S", and the error state is red with label "E". A dashed box below a transition indicates the number of the corresponding message (or reply) in the UML diagram. The PMC with id **o** models the behavior of the outermost sequence diagram in Figure 3.2, abstracting the behavior of the fragments associated with the features `memory` and `sqlite`. The behavior of these abstracted fragments is denoted by the PMCs with ids **m** and **s**, respectively.

The transitions of the PMC **o** that are parameterized using functions of the variable m (resp. s) abstract the reliability of the PMC **m** (resp. **s**), whenever the corresponding presence condition is satisfied. Otherwise, we assume the variable is set to 1, since an absent behavior cannot introduce an error. The parameterized transitions induce the dependency relation depicted by the green dashed arrows pointing from the dependent to the dependency. We name variables according to the id of the abstracted PMCs to ensure

correspondence. Figure 3.3 also depicts a simplified view of the PMC corresponding to the top-level UML behavioral model (i.e., the one that is not nested into any other). This PMC is labeled **root** and is present in all products.

3.2.1.1 Feature-family-based Reliability Analysis

ReAna is a Java-based tool, whose source code is free and publicly available[1]. The reliability analysis performed by ReAna follows the feature-family-based strategy: it consists of a feature-based analysis followed by a family-based analysis, and the analysis results of the feature-based analysis are used in the product-based analysis [Thüm et al., 2014]. Figure 3.4 is an outline of our particular approach, which can be summarized by the following steps:

1. Feature-based phase:
   (a) Each of the PMCs that result from model transformation (denoted by squares P_i) is model-checked using the parametric model checker PARAM [Hahn et al., 2010], resulting in a corresponding expression ε_i (represented by white circles). This parametric model checking step is denoted by α_v, because it is a variability-aware analysis.
2. Family-based phase:
   (a) Each expression ε_i, which expects that its variables be evaluated with Real values, is *lifted* to a corresponding ADD-based expression $\hat{\varepsilon}_i$.
   (b) The lifted expressions are evaluated in a bottom-up fashion (determined by a topological sort of the dependency graph) using the reliability ADD for each of its dependencies. The base case for this computation are constant expressions (corresponding to the innermost nested fragments in the behavioral diagrams). The evaluation function is denoted by σ_v, because we consider that the evaluation of lifted expressions is a variability-aware evaluation.

Each reliability ADD is depicted by a set of green circles (each representing a configuration) mapped (using a large arrow) to a set of yellow squares (each representing the computed reliability for a given configuration). The result is the topmost ADD, that is, the one corresponding to an expression on which no other depends. Such an expression corresponds to the top-level behavioral diagram—either an activity diagram or a sequence diagram that is not nested into another one—, which we call a *root*.

[1] https://github.com/SPLMC/reana

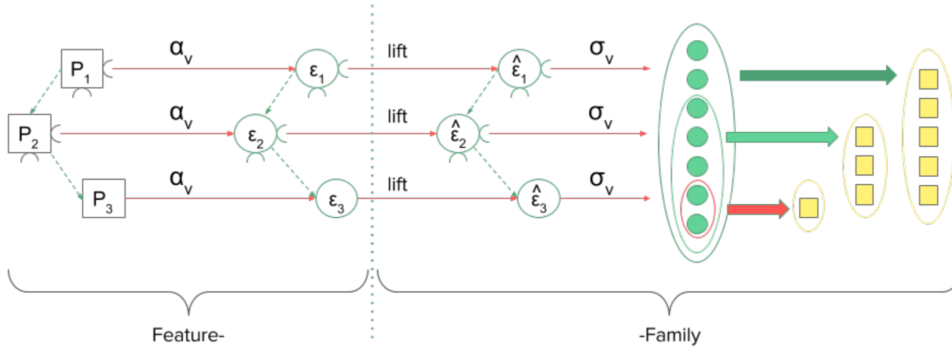


Figure 3.4: Outline of the implemented feature-family-based analysis approach. At the right side, we denote the resulting ADDs: sets of green circles (each denoting a configuration) mapped to sets of yellow squares (each denoting the corresponding reliability values—"products").

The process of lifting consists of interpreting Real constants in expressions as constant ADDs, arithmetic operators as ADD operators, and variables as ADD-typed variables. Formally, this means the acceptable evaluations for lifted expressions are of type $X \to (\mathbb{B}^k \to \mathbb{R})$, where X is the set of variables in the expression and $\mathbb{B}^k \to \mathbb{R}$ is the type of k-ary ADDs (k-ary Boolean functions to the Reals). In terms of implementation, the interpretation as Reals or as ADDs is performed at parsing time, meaning expressions are represented as strings until they need to be evaluated.

The first phase of the analysis consists of computing a reliability expression for each PMC in the behavioral model of the product line. Since this step is performed independently for each PMC, and since these PMCs correspond to the variation units in the original UML models (sequence diagram fragments—see Section 3.2.1), the parametric model checking of all PMCs makes up a feature-based phase. This computation only needs to be performed once, but the expressions remain to be evaluated.

To evaluate the expressions and thus check the reliability of the product line, we leverage ADDs as variational data structures [Walkingshaw et al., 2014]. The goal is to save processing time by storing reliability values for valid configurations in data structures that provide for efficient arithmetics. An ADD is handled as a function that maps a configuration of the product line into a corresponding reliability value. These mappings rely on the feature model's rules and on presence conditions for the behaviors corresponding to each PMC, effectively encoding the variability of the product line. Thus, this last phase of the analysis is family-based.

3.2.2 Product-line Extraction

But what would have happened if we chose not to use ADDs in the last phase of ReAna's original workflow? In this case, the expressions would not have to be recomputed, since the PMCs would have remained the same. However, the bottom-up computation of reliability values would have to be performed once for every configuration, evaluating the expressions with Real values according to the satisfaction of the respective presence conditions. This alternative second phase is product-based.

3.2.2.1 Feature-product-based Reliability Analysis

To accomodate the product-based alternative for evaluating the expressions, we cloned the class that was responsible for coordinating all analysis steps, and then refactored the last part of its analysis algorithm. The resulting analysis is feature-product-based, and it reuses the feature-based phase of the feature-family-based original analysis.

Figure 3.5 is an outline of our feature-product-based approach, which can be summarized by the following steps:

1. Feature-based phase:

    (a) Each of the PMCs that result from model transformation (denoted by P_i) is model-checked using the parametric model checker PARAM [Hahn et al., 2010], resulting in a corresponding expression ε_i. Again, this parametric model checking step is denoted by α_v, because it is a variability-aware analysis.

2. Product-based phase:

    (a) For every valid configuration of the product line, the expressions ε_i are evaluated in a bottom-up fashion (determined by a topological sort of the dependency graph) using the reliabilities computed by evaluating each of its dependencies and yielding reliabilities r_i (yellow squares). The base case for this computation are constant expressions (corresponding to the innermost nested fragments in the behavioral diagrams). The evaluation function is denoted by σ.

The iteration over valid configurations $c \in [\![FM]\!]$ is denoted by the green box, whose content quantifies over configurations, and the associated green frame, whose content is the actual iteration step. Each reliability value (the result from evaluating an expression) is denoted by r_i, corresponding to the expression ε_i from which it was computed. The resulting reliability of a product is the topmost value, that is, the one corresponding to an

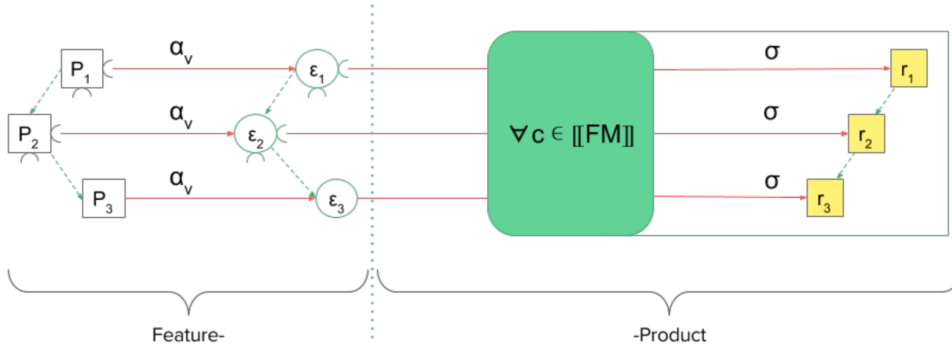


Figure 3.5: Outline of the implemented feature-product-based analysis approach. At the right side, we denote the resulting reliabilities by yellow squares. The green box denotes iteration over all valid configurations $c \in [\![FM]\!]$.

expression on which no other depends. As in the feature-family-based case, this expression corresponds to the root (top-level) behavioral diagram.

At this point, both feature-family-based and feature-product-based strategies were supported. This represented the first introduced variability, and thus we extracted the first version of the product line of analysis tools, *ReAna-SPL*.

It is worth noting that this feature-product-based approach resembles the one proposed by Ghezzi and Molzam Sharifloo [2013]. This is by no means a coincidence, since their work is closely related to the ongoing work by our research group on feature-family-based reliability analysis of product lines. This way, ReAna's original approach can be seen as a variation of the one by Ghezzi and Molzam Sharifloo.

3.2.3 Reactive Evolution

After implementing the feature-family-based and feature-product-based analysis strategies, we applied the extractive approach to product-line adoption to obtain the first version of ReAna-SPL. We then relied on an existing literature survey [Thüm et al., 2014] and on related work to extended our product line to support other strategies in the product-line analysis taxonomy [Thüm et al., 2014]. The input models and outputs remained the same throughout the process, so variability continued to be introduced as analyzers implementing different strategies.

3.2.3.1 Product-based Reliability Analysis

Ghezzi and Molzam Sharifloo [2013] presented a feature-product-based approach to reliability analysis of product lines, based on PMCs derived from annotated UML diagrams. They presented their technique as an alternative to the generation of a Markov model for each configuration—a product-based approach. Because of the similarity between their

input models and ours, we introduced a product-based strategy for ReAna-SPL that is based on their idea of a product-based analysis. However, whereas Ghezzi and Molzam Sharifloo [2013] resolve the variability of UML diagrams (yielding UML diagrams of products), we perform this variability binding on the derived PMCs.

Our product-based analysis strategy is outlined in Figure 3.6. For every valid configuration, we generate the corresponding probabilistic behavioral model (a DTMC) by *composing* the PMCs whose presence conditions are satisfied. This composition process consists of "inlining" a PMC in the places where the transfomation process created a variable to abstract it. Take, for instance, our example sequence diagram in Figure 3.2. Let x be the variable created in the PMC for the enclosing diagram at the point where the behavior conditioned by *memory* would be. Composition puts the PMC for *memory* as a replacement for the transition x, that is, at the point where it would be present if our transformation process did not split the behavior of different features.

Similar to expression evaluation, PMC composition is performed in a bottom-up fashion, using constant PMCs (i.e., DTMCs) as base cases. After this process of composition (which we name λ) is finished, the result is a DTMC modeling the behavior of the product corresponding to the configuration at hand. Then, we apply non-parametric model checking (denoted by α) to compute the reliability of this resulting model—also using the PARAM model checker.

Figure 3.6: Outline of the implemented product-based analysis approach. Nested squares represent DTMCs that result from composition.

3.2.3.2 Family-based Reliability Analysis

According to the analysis taxonomy [Thüm et al., 2014], a family-based strategy operates only in domain artifacts and incorporates the knowledge about valid feature combinations. In annotation-based product lines, this can be done using variability-aware analyzers that rely on variability-aware parsers [Kästner et al., 2011], for instance. Another technique is to merge all variability into a single *product simulator*, or *metaproduct*, which can be analyzed using off-the-shelf tools [Apel et al., 2011, 2013b; von Rhein et al., 2016]. We based our implementation on the latter technique, because our input models are

compositional—behavior modules that are meant to be composed with each other to yield the behavior of a given product. We also borrow the idea of conditioning transitions on feature presence from featured transition systems [Classen et al., 2013], although our concrete models and techniques differ.

Our family-based approach consists of encoding the variability of the whole product line in a single PMC. This process, which is an instance of *variability encoding* [Post and Sinz, 2008; von Rhein et al., 2016], is similar to PMC composition. However, we skip checking for satisfaction of presence conditions; instead, we perform the inlining of PMCs while still retaining the variables. The variable transitions of the resulting variability-encoded PMC have the semantics of choice: the system makes the transition to a state belonging to a given feature's behavior whenever the corresponding variable evaluates to 1 (*true*); otherwise, it skips all these states, to model the absence of the feature. This variability-encoded model can be seen as an asset with annotation-based variability representation.

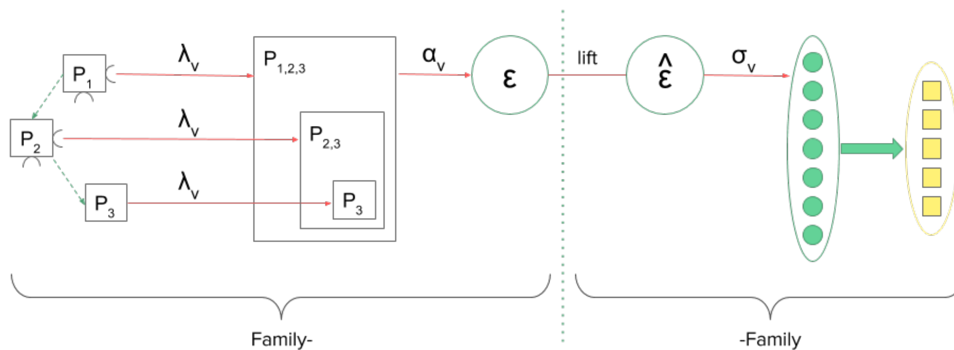


Figure 3.7: Outline of the implemented family-based analysis approach. Nested squares represent variability-encoded PMCs, with the outermost square denoting the result from embedding the variability of the whole product line into the root PMC. At the extreme right, we see a single ADD that maps all valid configurations to their respective reliabilities (cf. Figure 3.4).

Figure 3.7 is an outline of our family-based analysis. We first generate a variability-encoded PMC that is able to reproduce the behavior model of any product. This variability encoding process is abstracted by the function λ_v, named this way to suggest a "variability-aware derivation". Then, we apply parametric model checking to obtain a corresponding reliability expression. Unlike the expressions obtained from the individual compositional PMCs, the expression obtained from the variability-encoded model must be evaluated with Boolean values 0 and 1. This reflects the change in the semantics of variables that was introduced by variability encoding: instead of representing reliability values, they now represent feature presence or absence.

The resulting expression is lifted and then evaluated using variability-aware evaluation (σ_v), as in the family-based phase of the feature-family-based strategy. Each variable is evaluated using the product between the ADD for its corresponding presence condition and the ADD for the feature model's rules. This ensures the variable is only evaluated to 1 for *valid* configurations in which its presence condition is *satisfied*.

3.2.3.3 Family-product-based Reliability Analysis

Since the last part of the family-based strategy is similar to the family-based phase of the feature-family-based strategy, it enables a similar choice. Instead of lifting the expression and performing variability-aware evaluation, we can evaluate the expression once for each valid configuration. The resulting analysis is family-product-based, and is depicted by Figure 3.8. In the implementation, the common part (variability encoding followed by parametric model checking) is reused in both the family-based and the family-product-based strategies.

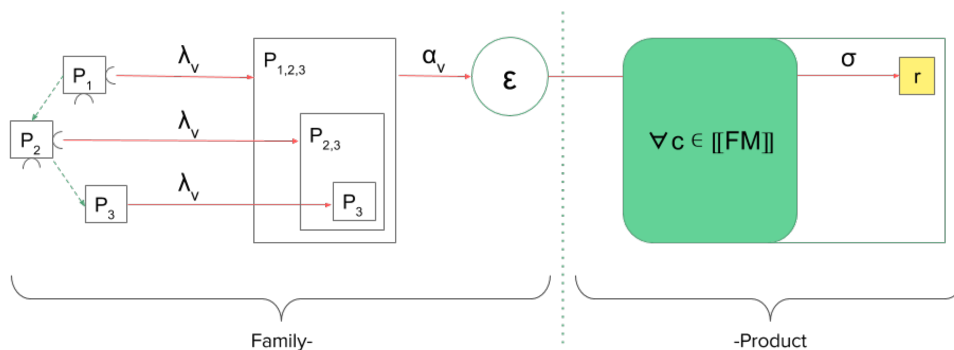


Figure 3.8: Outline of the implemented family-product-based analysis approach. In contrast with Figure 3.7, the result is represented by a reliability value (yellow box), independently computed for each valid configuration.

3.3 Product Line of Product-line Reliability Analysis Tools

By investigating patterns and similarities between the implemented strategies, we devised the diagram in Figure 3.9[2]. This diagram presents the implemented strategies as alternatives, each of which being a composition of intermediate analysis steps that act as building blocks. Starting at the top-left node (which represents the compositional model obtained from the initial transformation of UML models into PMCs), the reader can follow the

[2] This figure represents a preliminary version of Figure 4.7.

arrows to obtain the outline of an analysis technique. The strategy at hand is determined by the colors of the arrows, according to the legend at the bottom of the figure.

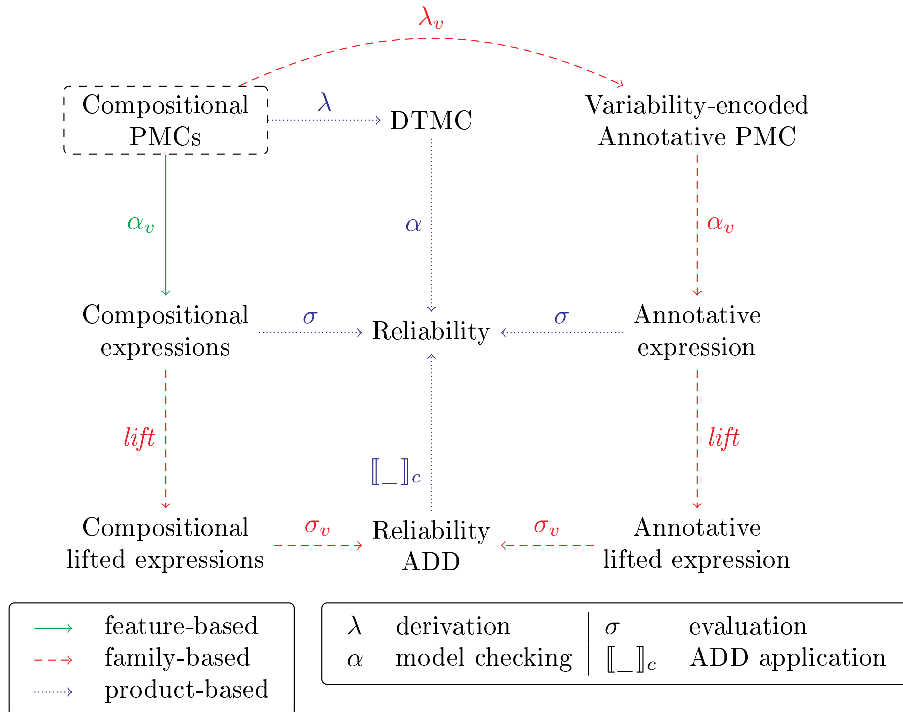


Figure 3.9: Outline of the relation between the implemented product-line reliability analysis strategies.

For instance, ReAna's original strategy consists of the following steps: start with a compositional model, use parametric model checking to obtain compositional expressions (solid green arrow labeled α_v), lift the resulting expressions to work with ADD semantics (dashed red arrow labeled *lift*), and then evaluate the resulting expressions (dashed red arrow labeled σ_v). Since the arrows in the path just described are (in order of appearance) solid green (feature-based step), dashed red (family-based step), and dashed red, the strategy is feature-family-based.

Note that, although *Reliability* is the only sink node in the diagram, we considered the above path to be complete when we reached *Reliability ADD*. The reason is that, for practical purposes, the reliability ADD contains the same information as would be given by a mapping from configuration values to reliabilities. Indeed, under the convention that a configuration is denoted by a Boolean tuple (Section 2.1), $[\![FM]\!] \subseteq \mathbb{B}^k$ for a product line with k features. Thus, both data structures have type $\mathbb{B}^k \to \mathbb{R}$.

The implemented product line, ReAna-SPL, is publicly available as free and open-source software[3]. It is a Java program with load-time variability bound by command-line arguments. The design decision to use load-time variability aimed at providing a tool that

[3] https://github.com/SPLMC/reana-spl

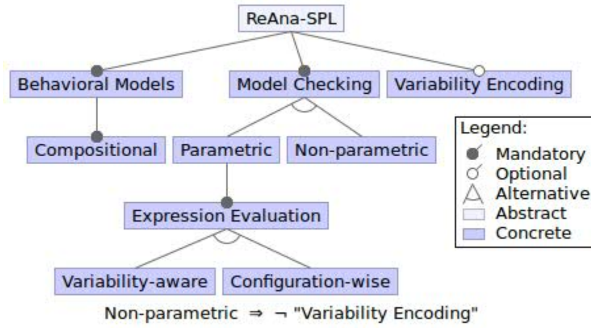


Figure 3.10: Feature model of the reliability analysis software product line (ReAna-SPL).

could be useful not only to perform a single type of analysis, but also to evaluate different analysis strategies in a timely fashion—e.g., performing a series of analysis experiments to empirically compare strategies.

Implementation Summary

Mechanism: *strategy* design pattern;

Binding time: load time;

Technology: language-based;

Representation: composition-based.

ReAna-SPL's feature model is shown in Figure 3.10. Each of its five valid configurations correspond to one of the possible paths in the diagram of Figure 3.9, and, therefore, to one of the implemented strategies. Table 3.1 presents the correspondence between valid configurations, analysis strategies, and the combined analysis steps. Each sequence of analysis steps denotes a path in the diagram in Figure 3.9. The `--strategy` command-line option switches between analysis strategies at load time. The strategies map to the accepted values by capitalizing all letters and replacing the hyphen with an underscore, that is, `FEATURE_FAMILY` corresponds to the feature-family-based strategy, `FAMILY` corresponds to the family-based strategy, and so on.

In ReAna-SPL, variability is realized by means of the *strategy* object-oriented design pattern [Gamma et al., 1995], in which different analysis strategies are implemented by classes in the *concrete strategy* stereotype (coarse-grained variability). The shared analysis phases were implemented using an additional level of the strategy pattern, and intermediate analysis steps were realized by functions. Design patterns provide good-practice

Configuration				Strategy	Analysis steps
Behavioral Models	Model Checking	Expression Evaluation	Variability Encoding		
Compositional	Non-parametric	-	`False`	Product-based	$\lambda \to \alpha$
Compositional	Parametric	Variability-aware	`False`	Feature-family-based	$\alpha_v \to lift \to \sigma_v$
Compositional	Parametric	Configuration-wise	`False`	Feature-product-based	$\alpha_v \to \sigma$
Compositional	Parametric	Variability-aware	`True`	Family-based	$\lambda_v \to \alpha_v \to lift \to \sigma_v$
Compositional	Parametric	Configuration-wise	`True`	Family-product-based	$\lambda_v \to \alpha_v \to \sigma$

Table 3.1: Correspondence between valid configurations, analysis strategies, and analysis steps.

guidelines for disciplined implementations of variability, as well as a means to decouple and encapsulate features [Apel et al., 2013a].

3.3.1 Quality Assessment

Despite the aforementioned advantage, using design patterns as a variability implementation mechanism also has weaknesses [Apel et al., 2013a]. However, they were manageable in our case. We now present the trade-offs of our product line, following the quality criteria suggested by Apel et al. [2013a] and a discussion of strong and weak points of using design patterns in general, also by Apel et al. [2013a].

Since we built ReAna-SPL using extractive and reactive adoption strategies, **preplanning effort** was low. Our progressive analysis of the domain of reliability analysis strategies suggested that the most common variation was the choice between strategies. This motivated the choice to use the *strategy* design pattern, leading to coarse **granularity**. A disadvantage is that it restricts the ways in which features can be combined to support new configurations, since features combinations are *hard-coded* as concrete strategies. Nonetheless, since the analyses are implemented in terms of lower-level building blocks (representing intermediate analysis steps), we expect the effort to implement a new analysis strategy to be reduced.

Feature traceability is not immediate, since ReAna-SPL's source code is structured in terms of analysis strategies, whereas its feature model (Figure 3.10) describes finer-grained domain concepts. However, we address this issue using clear mappings from feature selections to analysis strategies (Table 3.1) and from the latter to analyzer classes (concrete strategies) in the source code. Furthermore, intermediate analysis steps, which are functions used to implement the strategies, are traceable to features (see Table 3.1).

Separation of concerns is addressed by having strategy classes that are only responsible for coordinating an analysis, delegating intermediate steps to cohesive functions. The same design provides for **information hiding**, whereby analysis logic is properly encapsulated behind strategy interfaces. Since we only deal with variability in Java code using the *strategy* pattern, our variability mechanism is trivially **uniform**.

The fact that code for all variants is deployed did not present a space problem, since code for UML parsing, models transformation, model checking, and run-time statistics gathering, which belongs to the base code, accounts for approximately 58% of ReAna-SPL's source code[4]. Moreover, we did not find run-time defects related to coexisting variant logic. Boilerplate code and architectural overhead were also not an issue. The model checking and variability-aware expression evaluation steps represent the performance bottlenecks during analysis, so that the performance penalty of strategy dispatching and calls to reused functions is negligible.

Quality Assessment

Preplanning effort: low (extractive and reactive adoption strategy);

Feature traceability: not immediate, but addressed by Table 3.1;

Separation of concerns: high cohesion;

Information hiding: strategy interfaces and encapsulated analysis logic;

Granularity: coarse-grained changes (concrete strategies);

Uniformity: the only assets are Java classes, and only design patterns are exploited to implement variability.

3.3.2 Empirical Validation

Before we set out to formalize the implemented strategies and demonstrate their soundness, we performed an empirical investigation to gather evidence that they, indeed, commute. This experiment consisted of analyzing behavioral models of six product lines using all of our strategies. For each product line, the results from the product-based analysis were taken as a baseline, to which the results obtained using the other strategies were compared configuration-wise.

Our purpose was not to measure performance (neither in time nor in space), but to assess whether all strategies yield reliabilities that can be deemed equal. Moreover, the choice to use product-based analyses as baselines does not imply that this analysis strategy is more precise. This decision stems from the fact that the product-based strategy

[4] This estimate was obtained by counting lines of code in the packages that are responsible for the aforementioned common tasks and comparing to the total count of lines of code in ReAna-SPL (excluding unit tests). The counts ignored comments and blank lines, and were performed using the CLOC tool (http://cloc.sourceforge.net/).

performs a single-product analysis for each product of the product line, thus corresponding to a regular (i.e., not tailored to software product lines) software analysis method. Therefore, this strategy is sound iff the underlying single-product analysis is sound.

Table 3.2 shows the subject product lines, along with the corresponding number of features and size of the configuration space. These product lines were chosen due to the availability of their variability model, but also because they were being subject to other empirical studies within our research group. Additionally, *EMail*, *MinePump*, *BSN*, and *Lift* were used in previous work addressing model checking of product lines. The behavioral models and feature models of all subject systems are available in ReAna-SPL's source code repository (`https://github.com/SPLMC/reana-spl`).

	Features	**Configurations**
EMail [University of Magdeburg]	10	40
MinePump [Kramer et al., 1983]	11	128
BSN [Rodrigues et al., 2015]	16	298
Lift [Plath and Ryan, 2001]	10	512
InterCloud [Ferreira Leite et al., 2015]	54	110592
TankWar [University of Magdeburg]	144	4.21×10^{18}

Table 3.2: Product lines used for empirical validation.

Since expression evaluation relies on floating-point operations, we compared results using relative errors [Goldberg, 1991]. If r_{0_c} is the reliability computed for configuration c using the product-based strategy (i.e., the baseline for comparison), and if r_c is the reliability obtained for the same configuration c using another given strategy, the relative error is given by

$$err = \frac{|r_{0_c} - r_c|}{r_{0_c}}$$

	Maximum Relative Error			
	Feature-family-based	**Feature-product-based**	**Family-based**	**Family-product-based**
EMail	4.37×10^{-16}	4.37×10^{-16}	2.19×10^{-16}	2.19×10^{-16}
MinePump	1.34×10^{-15}	1.34×10^{-15}	3.71×10^{-16}	3.71×10^{-16}
BSN	2.15×10^{-16}	2.15×10^{-16}	2.07×10^{-16}	2.07×10^{-16}
Lift	5.33×10^{-16}	5.33×10^{-16}	3.26×10^{-16}	3.26×10^{-16}
InterCloud	2.14×10^{-16}	2.14×10^{-16}	*out of memory*	*out of memory*
TankWar	*no baseline available*			

Table 3.3: Maximum relative errors for each analysis strategy, using the product-based analysis as a baseline.

Table 3.3 shows the maximum relative error of applying each strategy to a subject product line. The product-based strategy does not appear in this table, since it was used as a baseline for computing the errors. All errors where contained within a margin of approximately 10^{-15}, with most of them within approximately 10^{-16}. Since we have implemented expression evaluation using 64-bit double-precision floating-point numbers

(the `double` data type), and since the relative errors in Table 3.3 are close to the unit roundoff for this data type ($\approx 1.11 \times 10^{-16}$ [Higham, 2002]), we believe the errors are due to rounding in floating-point arithmetics.

To raise the level of confidence in this conclusion, we compared the corresponding relative errors (i.e., configuration-wise) for each pair of strategies. With this comparison, we found out that the relative errors of the results obtained by the feature-family-based and by the feature-product-based strategies are equal. This is consistent with the fact that both feature-based strategies perform arithmetics over the same set of expressions (each corresponding to a PMC in the behavioral model), and thus perform the same operations over the same `double` values. The same happens with the family-based and family-product-based strategies: both consist of evaluating the same expression (obtained from parametric model checking of the variability-encoded PMC), and have correspondingly equal relative errors. These results provide empirical evidence that the implemented analysis strategies commute, in the sense that they are equivalent to one another. Nonetheless, further empirical studies can strengthen this evidence by repeating the comparison for a greater number of product lines.

Due to the large configuration space, a product-based analysis of the TankWar product line was not practical, so it was not evaluated. Indeed, the only implemented strategy that managed to analyze this product line in our computing environment was the feature-family-based strategy. Enumerative strategies (product-based and feature-product-based) timed out, while the ones based on variability encoding (family-based and family-product-based) ran out of memory. While analyzing the InterCloud product line, family-based and family-product-based strategies also ran out of memory. Future work should investigate the reasons for this behavior, as well as the absolute and relative performance of each analysis strategy.

3.4 Theory Development

After implementing the five preceding analysis strategies in ReAna-SPL, we started seeking for recurring patterns. To this end, we compared the source code of the analyzers with one another, searching for redundancies and similar programming techniques.

Coarse-grained patterns related to feature-based and family-based first phases were the first to be noticed, since they occurred by design. In Figures 3.4 and 3.5, for instance, we see that the outline of the feature-based first phase is the same in both strategies. The same happens in the family-based first phases of the family-based (Figure 3.7) and the family-product-based (Figure 3.8) strategies.

Comparing the second phases, however, unintended patterns start to emerge. Take Figures 3.4 and 3.7, for instance. The outlines of the family-based second phases of the feature-family-based and the purely family-based strategies have similar shapes. Both rely on lifting and variability-aware evaluation, but they do so in different ways: whereas the family-based strategy handles a single expression whose variables are evaluated as presence values (0 or 1), the feature-family-based strategy evaluates a number of interdependent expressions in a bottom-up way, using the resulting reliability values (Real numbers in the $[0, 1]$ interval) to evaluate variables. Hence, the variability-aware evaluation function σ_v can be broken down into a mapping from configurations to evaluations (an "evaluation factory") and a combinator that turns this mapping into a concrete expression evaluator. Evaluating in the feature-family-based scenario can thus be seen as a *fold* (or *reduce*) combinator applied to a list of lifted expressions and a generic variability-aware evaluation combinator σ_v. A similar pattern occurs with the evaluation function σ in the product-based second phase of the feature-product-based strategy (Figure 3.5) and the derivation function λ in the purely product-based strategy (Figure 3.6).

Another pattern arises when comparing derivation and variability encoding. Both transformations rely on a PMC composition mechanism, but, whereas derivation rules out variables by evaluating presence conditions, variability encoding wraps the composed PMCs with parametric transitions that model configuration choice as a behavior. Figure 3.11 illustrates these differences and similarities. In this figure, probabilities of transitions that do not participate in the transformation at hand were ommited for brevity.

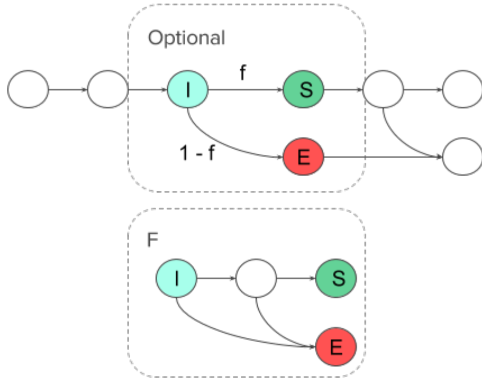
The encoding of configuration-time choices as conditional constructs (*if-then-else* operators) is also recurring. This pattern, which was already present in works on variability encoding of source-code assets [Apel et al., 2011; Post and Sinz, 2008; von Rhein et al., 2016], was identified not only within variability encoding of PMCs. It also arises when performing variability-aware evaluation (by means of the `ITE` ADD operator). Table 3.4 summarizes the occurrences of the *if-then-else* pattern, specifying what construct is used to switch on which conditional, as well as the corresponding consequent processing (which takes place if the conditional is *true*) and the alternative value (used whenever the conditional is *false*). More details on the pattern instances are provided in Section 4.2.

	If-then-else	**Conditional**	**Consequent**	**Alternative**
Expression evaluation (σ)	*if* statement	Presence condition satisfaction	Evaluate with $\mathbb{R}$ semantics	1
Variability-aware expression evaluation (σ_v)	ADD `ITE` operator	Presence condition ADD	Evaluate with `ADD` semantics	**1** (constant ADD)
DTMC derivation (λ)	*if* statement	Presence condition satisfaction	Compose	Trivial PMC†
Variability encoding of PMCs (λ_v)	PMC `ITE` operator	PMC identifier	Compose	Trivial PMC†

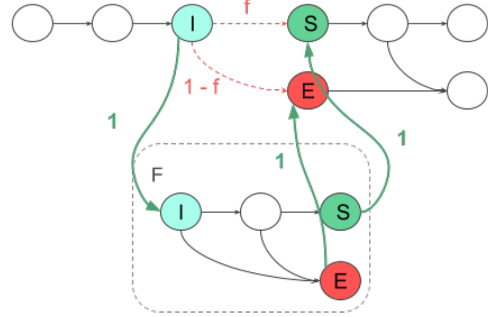
† Composing the trivial PMC does not affect reachability probabilities (see Section 4.1.2).

Table 3.4: Occurrences of the *if-then-else* pattern.

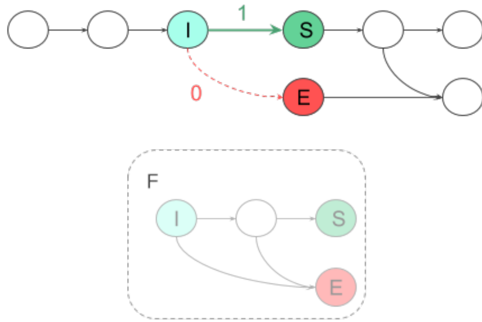
As a visual representation of the implemented product-line and its building blocks, the diagram in Figure 3.9 also enabled the discovery of new possibilities. For instance, we



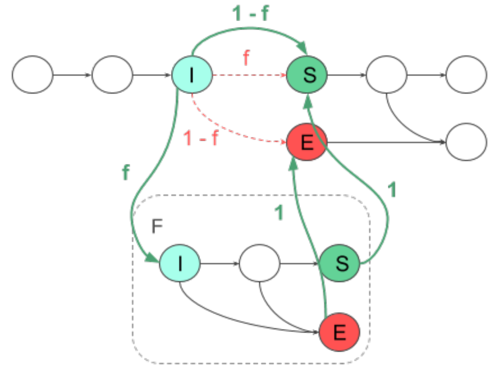
(a) Base PMC and a component PMC conditioned on feature F. Variable f is used to abstract this component's behavior.



(b) Result of derivation (λ) if feature F is present.



(c) Result of derivation (λ) if feature F is not present.



(d) Result of variability encoding (λ_v). The added transitions (green arrows) seem to wrap the component PMC with an *if-then-else* conditional.

Figure 3.11: Comparison of derivation and variability encoding. Green solid arrows denote new transitions, and dashed red arrows represent transitions that are removed by the transformation process.

saw an opportunity to provide a product-based step to derive annotation-based models, yielding an alternative product-based approach. Using the unveiled patterns and the diagram in Figure 3.9 as a guidance, we formalized the probabilistic models targeted by our analysis strategies (Section 4.1), as well as the implemented and discovered strategies themselves (Section 4.2).

With the domain knowledge acquired in this formalization phase, our feature model evolved to the one presented in Figure 3.12. This new feature model represents the variability in the supported representations of variability in models (either compositional or annotative). Moreover, the cross-tree constraint stating that `Non-parametric` $\Rightarrow$ $\neg$`Variability Encoding` was dropped. It reflected the previous lack of product-based analysis of annotative models, a limitation of the implemented tool that was

overcome by the theoretical results. Furthermore, we added the constraint `Annotative` ⇒ ¬`Variability Encoding`, representing the fact that variability encoding has no effect over our annotative models.

Table 3.5 presents the correspondence between valid configurations, strategies, and analysis steps after evolution. For compositional models, the difference is that variability encoding of such models is now allowed to coexist with non-parametric model checking. For annotative models, all configurations are new, since the `Annotative` feature did not exist in the previous version. However, the sequence of analysis steps of each of these configurations corresponds to a suffix of the sequence of analysis steps of one of the three configurations based on variability encoding of compositional models. This happens because such variability encoding is effectively a model translation. Thus, all analyses that encode variability of compositional models are also valid when applied directly to annotative models.

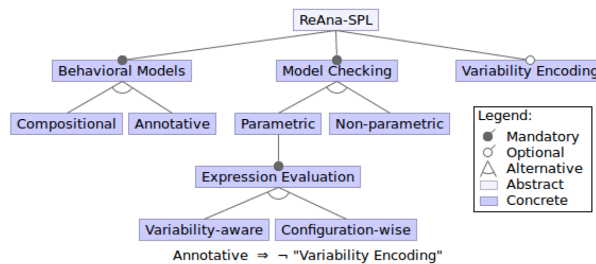


Figure 3.12: Feature model of the reliability analysis software product line (ReAna-SPL) after feedback from the formalization phase (Chapter 4).

Configuration				Strategy	Analysis steps
Behavioral Models	Model Checking	Expression Evaluation	Variability Encoding		
Compositional	Non-parametric	-	`False`	Product-based	$\lambda \to \alpha$
Compositional	Non-parametric	-	`True`	Family-product-based	$\lambda_v \to \lambda \to \alpha$
Compositional	Parametric	Variability-aware	`False`	Feature-family-based	$\alpha_v \to lift \to \sigma_v$
Compositional	Parametric	Configuration-wise	`False`	Feature-product-based	$\alpha_v \to \sigma$
Compositional	Parametric	Variability-aware	`True`	Family-based	$\lambda_v \to \alpha_v \to lift \to \sigma_v$
Compositional	Parametric	Configuration-wise	`True`	Family-product-based	$\lambda_v \to \alpha_v \to \sigma$
Annotative	Non-parametric	-	`False`	Product-based	$\lambda \to \alpha$
Annotative	Parametric	Variability-aware	`False`	Family-based	$\alpha_v \to lift \to \sigma_v$
Annotative	Parametric	Configuration-wise	`False`	Family-product-based	$\alpha_v \to \sigma$

Table 3.5: Correspondence between valid configurations, analysis strategies, and analysis steps after formalization of strategies.

We also demonstrated that the family-based intermediate analysis steps at the right side of Figure 3.9 commute, thereby proving the soundness of our family-based and family-product-based analysis techniques (after the variability encoding step). This way, we established a valid relation between product-line reliability analysis strategies. The resulting definitions, lemmas, and theorems, along with the corresponding proofs, were subject to

scrutiny both within our group and by fellow researchers. This work is currently being prepared for submission to a peer-reviewed journal.

During the theory development process, we employed graphs to map dependencies between elements in our theory (i.e., definitions, lemmas, and theorems). These graphs, shown in Appendix B, informed on what elements would be impacted by a change to a given element. Thus, they helped in maintaining consistency and correctness throughout the rounds of external and internal review.

In the following chapters, we report the theoretical results of our work.

Chapter 4

Commuting Strategies for Product-line Reliability Analysis

This chapter presents the formalization of our behavioral models for software product lines (Section 4.1) and of our analysis strategies (Section 4.2). It also presents a formulation of the soundness of our strategies as theorems, along with corresponding proofs.

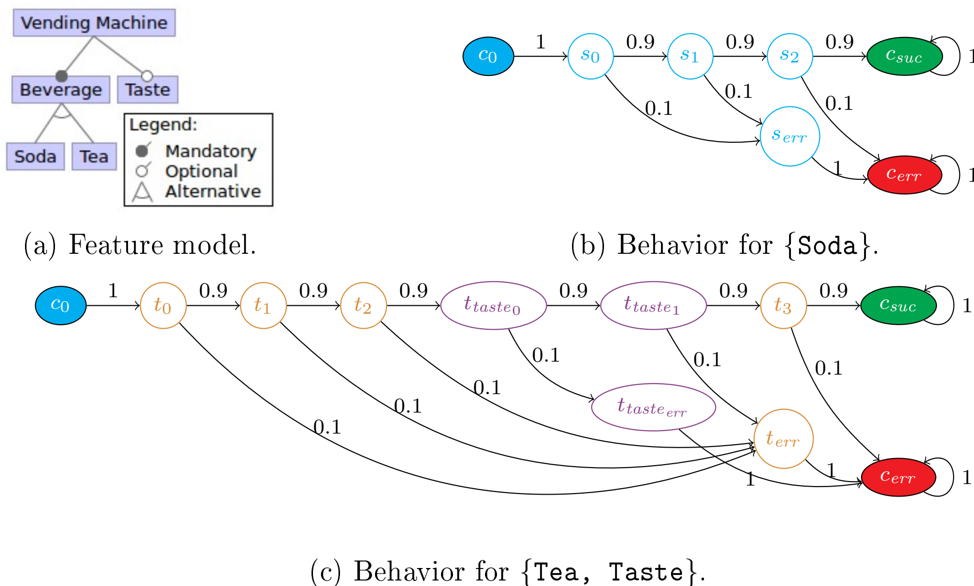
The discussion is focused on annotative models and analyses thereof, that is, the family-based and family-product-based strategies. Nonetheless, we also provide informal insights on the compositional models exploited by our feature-family-based (Section 3.2.1.1) and feature-product-based (Section 3.2.2.1) strategies. To better illustrate the formal concepts, we provide a running example.

4.1 Markov-chain Models of Product Lines

Reliability analysis, in our setting, is the application of probabilistic model checking to a probabilistic model of a software system. However, for a product line, it may not be feasible to manually model each product (i.e., its probabilistic model) and then analyze it, due to exponential blowup. Hence, we model the product line as a whole in terms of its common and variable behavior, to enable the automatic derivation of probabilistic models corresponding to the behavior of each product of the product line. Such variable behavioral models have properties that allow them to be used with different analysis strategies, as we will show in Section 4.2. Although we show and use precise definitions of the resulting models, it is outside the scope of this work to present modeling techniques to create them. Models can be produced, for example, by using behavioral UML diagrams annotated with component reliabilities [Ghezzi and Molzam Sharifloo, 2013; Nunes et al., 2013] or feature-oriented formalisms [Chrszon et al., 2016].

Since single-product analysis relies on DTMCs to model software behavior, we use PMCs to represent DTMC variability in product-line analysis. To illustrate our approaches to variability representation and product-line analysis, yet without loss of generality, we rely on an example product line of beverage vending machines (Figure 4.1), slightly modified from the examples in the work by Ghezzi and Molzam Sharifloo [2013] and Classen et al. [2010]. This product line consists of models of vending machines that are able to deliver tea or soda (but never both) and, for each case, there is a beverage-specific optional behavior of adding taste.

The feature model for this product line is depicted in Figure 4.1a, where `Soda` and `Tea` are alternative features (i.e., they cannot be simultaneously present in a feature selection) representing the behaviors of serving soda and tea, respectively. Since adding taste to a beverage is an optional behavior, it is modeled by the optional feature `Taste`. If a product is generated with the feature selection {`Soda`} (i.e., `Taste` is not selected), a possible model of its probabilistic behavior is depicted in Figure 4.1b. If the feature selection is {`Tea, Taste`}, the derived product has a probabilistic behavioral model as in Figure 4.1c.



(a) Feature model. (b) Behavior for {`Soda`}.

(c) Behavior for {`Tea, Taste`}.

Figure 4.1: Vending machine product line example.

In both example DTMCs, transitions indicate a change in the machine's execution state, with probabilities representing the reliabilities of the corresponding execution steps. These reliabilities are usually taken to be the probabilities that the software components responsible for each step will successfully produce the expected outcome. In this sense, one can notice most states have two outgoing transitions: one representing success and another representing failure. The states with only one outgoing transition may be seen as execution control handoffs. Also, to help us identify variation points, states are labeled

according to the behavior they model and are correspondingly colored. Label c denotes *common* behavior (present in all products), while s and t denote behaviors introduced by features Soda, and Tea, respectively. States labeled t_{taste} correspond to the behavior of adding taste to tea, that is, they only exist in products derived by a feature selection with both features Tea and Taste.

As with source code, the way variability is represented as PMCs and the way products (i.e., DTMCs) are generated from the resulting variable assets can be classified in two main categories: annotation-based (or annotative) and composition-based (or compositional) [Apel et al., 2013a; Kästner et al., 2008]. We present the intuition for both kinds of variability representation and a formal definition of annotation-based models. The formalization of composition-based models is the subject of ongoing research.

4.1.1 Annotative Models

To represent the variable behavior of a product line in an annotative way, we use a PMC in which variables are interpreted as configuration-specific behavior selectors. Such a PMC for the vending machine product line is shown in Figure 4.2, where we introduce blue dashed states to represent configuration-specific behavior selection. For instance, to represent the variability for Tea-related behavior, we introduce a state labeled sel_t, which transitions to t_0 (not shown) with probability 1, if it is present, or transitions to the point right after the same behavior (a state correspondingly labeled aft_t) with probability 1, if it is absent. This mutually exclusive selection is represented by labeling transitions with the expressions t and $1 - t$, such that evaluating t as 1 yields the expected "present" behavior, while evaluating it with 0 yields the "absent" behavior. The same approach is also applied to the behavior corresponding to adding taste to tea. Some states of the model for serving tea, as well as the behaviors corresponding to Soda and its taste-adding variant, are omitted for brevity. The whole model can be seen in Figure A.1.

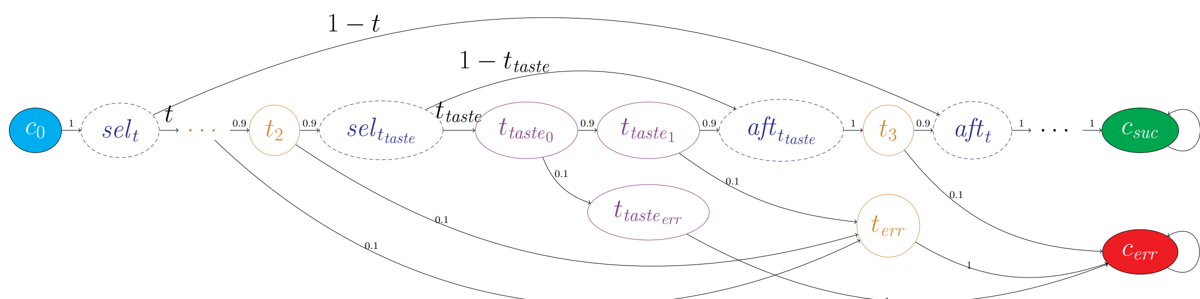


Figure 4.2: Annotative PMC for the vending machine.

We generalize and formally define this annotative approach of variability representation as follows.

Definition 4 (Annotative PMC)**.** An annotative PMC is a PMC $(S, s_0, X, \mathbf{P}, T)$ such that for all states $s \in S$, either:

1. $\forall_{s' \in Succ(s)} \cdot \mathbf{P}(s, s') \in [0, 1] \wedge \mathbf{P}(s, Succ(s)) = 1$ (the probabilities of all outgoing transitions are constants that add up to 1); or
2. $\exists_{s_1, s_2 \in S}\ \exists_{x \in X} \cdot Succ(s) = \{s_1, s_2\} \wedge \mathbf{P}(s, s_1) = x \wedge \mathbf{P}(s, s_2) = 1 - x$ (there are exactly two outgoing transitions, whose probabilities are expressed as a single variable and its complement).

The states in Figure 4.2 that fall in the second case are sel_t and $sel_{t_{taste}}$ (as well as sel_s and $sel_{s_{taste}}$, which are not shown), while all others fall in the first case. Each variable of an annotative PMC denotes the presence of a given behavior in a product. The intended semantics is that the sets of states and transitions giving rise to the denoted behavior will be reachable within the model if, and only if, its corresponding variable evaluates to 1.

For such an annotative PMC to represent the variable behavior of a product line with feature model FM, we must be able to use it to derive the behavioral model of any product generated by a configuration $c \in FM$. However, the use of a PMC by itself does not help with restricting the possible evaluations to achieve that. Evaluating the introduced variables with values other than 0 and 1 may yield ill-formed DTMCs (e.g., violating the stochastic property). Also, a variable should evaluate to 1 if, and only if, the presence condition of the subsystem whose behavior is controlled by this variable is satisfied. Hence, we need to constrain evaluations of this annotative PMC to reflect the corresponding feature model and presence conditions.

The first step towards this goal is to formalize what presence conditions mean in the context of variable behavior models. Thus, let p_x be the presence condition for the behavior identified by x. In our vending machine example, we would have $p_t = \texttt{Tea}$, $p_{t_{taste}} = \texttt{Tea} \wedge \texttt{Taste}$, $p_s = \texttt{Soda}$, and $p_{s_{taste}} = \texttt{Soda} \wedge \texttt{Taste}$. To precisely associate a variable to a presence condition, we define a higher-order function that maps a variable to a Boolean function over the features (see Section 2.1), which we call *presence function*.

Definition 5 (Presence function)**.** Given a set X of variables and a feature model FM, a presence function is a function $p : X \to (\llbracket FM \rrbracket \to \mathbb{B})$ such that, for all $x \in X$ and all $c \in \llbracket FM \rrbracket$,

$$p(x)(c) = \begin{cases} 1 & \text{if } c \models p_x \\ 0 & \text{otherwise} \end{cases}$$

where p_x is the presence condition associated with the variable x and $c \models p_x$ means that the configuration c *satisfies* p_x.

Next, we must be able to use the feature model to define evaluations. For instance, the annotative PMC for the vending machine product line would allow serving both tea and soda, if both t and s were evaluated to 1. However, this behavior is forbidden by the feature model, which states that `Tea` and `Soda` are alternative features. By incorporating knowledge of the feature model to evaluations, we can model all variant behavior as if it were optional and enforce the constraints of alternative and OR features when evaluating the PMC. The solution to this problem are higher-order functions complying to the following definition of an *evaluation factory*.

Definition 6 (Evaluation factory)**.** Given a feature model FM and a set X of variables, an evaluation factory $w : [\![FM]\!] \to (X \to \mathbb{R})$ is a function that, for a given configuration $c \in [\![FM]\!]$, yields an evaluation $w(c) \in X \to \mathbb{R}$.

At this point we have defined what we mean by an annotative PMC as well as an abstract means to constrain possible evaluations to the ones that make sense in the context of a given product line. For the particular case of annotative PMCs, an evaluation factory must generate evaluations that interpret variables as presence values and according to the presence conditions. Thus, we need to interpret the set $\{0, 1\}$ of *numbers* as the set $\mathbb{B}$ of Boolean values and restrict the generated evaluations to have this set as image. With this in mind, we define an *annotative probabilistic model* as follows:

Definition 7 (Annotative probabilistic model)**.** An annotative probabilistic model is a tuple $(\mathcal{P}, p, w, FM)$ such that:

- $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$ is an annotative PMC (Definition 4);
- FM is a feature model;
- $p : X \to ([\![FM]\!] \to \mathbb{B})$ is a presence function (Definition 5); and
- w is an evaluation factory such that, for all $c \in [\![FM]\!]$ and $x \in X$,

$$w(c)(x) = \begin{cases} 1 & \text{if } p(x)(c) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Remark 1 (Pointwise definition of w)*.* For practical purposes, it is worth noting that the right-hand sides of the definitions of w (Definition 7) and of the presence function p (Definition 5) are the same. That is, one can operationalize w as $w(c)(x) = p(x)(c)$, so the annotative evaluation factory could be uniquely determined from an annotative PMC

$\mathcal{P}$, a presence function p, and a feature model FM. Nonetheless, we keep w as part of the annotative model tuple to explicitly represent this configuration knowledge.

Starting with such an annotative model, the derivation of a specific behavioral model of a product with configuration $c \in [\![FM]\!]$ is then carried out by applying the evaluation $w(c)$ to the underlying PMC $\mathcal{P}$. Since PMC evaluation is not restricted to annotative PMCs, we define this process of DTMC derivation without resorting to the just defined concept of annotative models.

Definition 8 (DTMC derivation)**.** Given a PMC $(S, s_0, X, \mathbf{P}, T)$, a feature model FM, and an evaluation factory $w : [\![FM]\!] \to (X \to \mathbb{R})$, the DTMC derivation function $\lambda : PMC_X \times \big([\![FM]\!] \to (X \to \mathbb{R})\big) \times [\![FM]\!] \to DTMC$ is such that

$$\lambda(\mathcal{P}, w, c) = \mathcal{P}_{w(c)}$$

where PMC_X is the set of PMCs with variables set X. For brevity, we can also note $[\![\mathcal{P}]\!]^w_c$ to mean $\lambda(\mathcal{P}, w, c)$.

Note that the analysis methods we exploit in this work rely on evaluations being well-defined (Definition 3). This is where the restrictions we imposed on annotative models come into play: the evaluation factory of an annotative model always yields well-defined evaluations for the underlying annotative PMC.

Lemma 2 (Evaluation well-definedness for annotative models)**.** *For every annotative model $(\mathcal{P}, p, w, FM)$, $w(c)$ is a well-defined evaluation for $\mathcal{P}$, for all $c \in [\![FM]\!]$.*

Proof. By definition of well-defined evaluation for a PMC $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$ (Definition 3), an evaluation u is well-defined iff $\mathcal{P}_u$ obeys the stochastic property and $\mathbf{P}_u$ assigns a valid probability value to each transition. That is, $\forall_{s \in S} \cdot \mathbf{P}_u(s, Succ(s)) = 1$ and $\forall_{s,s' \in S} \cdot \mathbf{P}_u(s, s') \in [0, 1]$.

From Definition 7, $\mathcal{P}$ is an annotative PMC (Definition 4), so states with no variability (case 1) satisfy the needed properties by definition. For states s with variability (case 2), it holds that

$$\exists_{s_1, s_2 \in S}\ \exists_{x \in X} \cdot Succ(s) = \{s_1, s_2\} \land \mathbf{P}(s, s_1) = x \land \mathbf{P}(s, s_2) = 1 - x$$

Let us consider each property whenever $u = w(c)$:

Stochastic property. By definition,

$$
\begin{aligned}
\sum_{s' \in Succ(s)} \mathbf{P}_{w(c)}(s, s') &= \mathbf{P}_{w(c)}(s, s_1) + \mathbf{P}_{w(c)}(s, s_2) \\
&= \mathbf{P}(s, s_1)[X/w(c)] + \mathbf{P}(s, s_2)[X/w(c)] \\
&= x[X/w(c)] + (1 - x)[X/w(c)] \\
&= w(c)(x) + (1 - w(c)(x)) \\
&= 1
\end{aligned}
$$

Valid probabilities. From Definition 7, we have that for every $c \in [\![FM]\!]$, the image of $w(c)$ is $\{0, 1\} \subseteq [0, 1]$. Hence, either $\mathbf{P}_{w(c)}(s, s_1) = 1 \wedge \mathbf{P}_{w(c)}(s, s_2) = 0$ or $\mathbf{P}_{w(c)}(s, s_1) = 0 \wedge \mathbf{P}_{w(c)}(s, s_2) = 1$. That is, all possible transition probabilities lie in the $[0, 1]$ interval.

As there is no other case to consider, $\mathcal{P}_{w(c)}$ satisfies the required properties. Thus, $w(c)$ is well-defined for $\mathcal{P}$. □

In summary, an annotative probabilistic model represents all products of the product line, relying on presence conditions to define which parts have to be removed to derive a concrete product model. Because of that, this type of model is also known as 150% model [Haber et al., 2013], metaproduct [Thüm et al., 2013], variant simulator [von Rhein et al., 2016], or product simulator [Apel et al., 2011].

4.1.2 Compositional Models

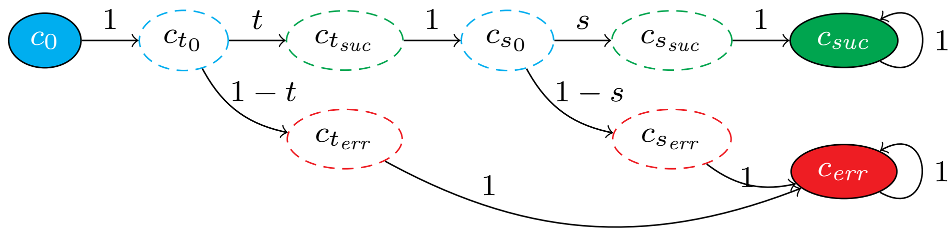
A compositional representation of variable configuration-specific behavior consists of a number of PMCs whose variables represent variation points, such that they can be composed with one another at predefined locations. To model a product line in this way, we start with a PMC comprising all common behavior, while abstracting all variable configuration-specific behavior. We then model each abstracted behavior as a DTMC, if it presents no further variability, or as another PMC, otherwise. In the latter case, we follow the same procedure to abstract inner variation points, until all behavior is modeled.

Figure 4.3 illustrates this concept. For the vending machine example, the *top-level* PMC $\mathcal{P}_\top$ would be as in Figure 4.3a. In this PMC, we introduce triples of dashed states that act as placeholders for the abstracted behavior. We call these states and corresponding transitions *slots*. For instance, the top-level PMC in Figure 4.3a has two slots, abstracting the behaviors of serving tea and soda. The tea slot consists of two elements: (a) the set of states c_{t_0}, $c_{t_{suc}}$, and $c_{t_{err}}$, representing the initial, success, and error states

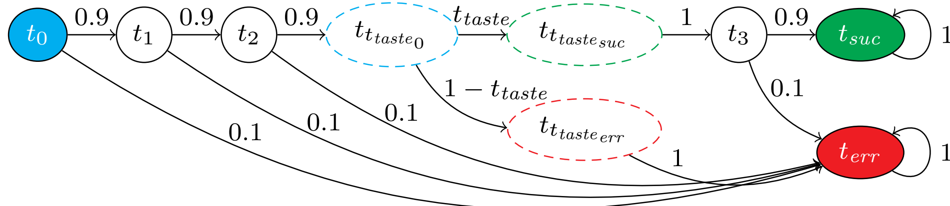
in the abstracted behavior, respectively; and (b) two transitions, annotated with the expressions t and $1-t$, denoting the probabilities of success and failure of this behavior, respectively. This way, we not only use the variable t as a slot identifier, but give it the possibility to be interpreted as the reliability of the tea behavior.

Note that, despite being alternatives, the behaviors of serving tea and soda are both represented in this PMC. This parametric model, by itself, does not prohibit the behavior of serving tea *and* soda subsequently. Like in the annotative representation of the vending machine (Figure 4.2), we do not enforce the rules of the feature model in the PMC itself. Instead, we ensure valid combinations of features during the composition process, as we shall see later.

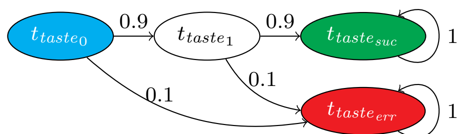
Figure 4.3b shows the PMC $\mathcal{P}_t$ for the tea behavior, in which we use a slot to abstract the optional taste-adding behavior, whose behavior is modeled by the PMC $\mathcal{P}_{t_{taste}}$ in Figure 4.3c. Since this tea-taste PMC has no variability, it is in fact a regular DTMC. We omit the PMCs for serving soda ($\mathcal{P}_s$) and for adding taste to soda ($\mathcal{P}_{s_{taste}}$), for brevity, but the complete example can be seen in Figure A.2 (Appendix A).



(a) Top-level compositional PMC $\mathcal{P}_\top$ for the vending machine (common behavior and main variation points).



(b) Compositional PMC $\mathcal{P}_t$ for the behavior of serving tea.



(c) Compositional PMC $\mathcal{P}_{t_{taste}}$ for the behavior of adding taste to tea.

Figure 4.3: Compositional PMCs for the vending machine.

The semantics of variables in a compositional parametric chain is different from the

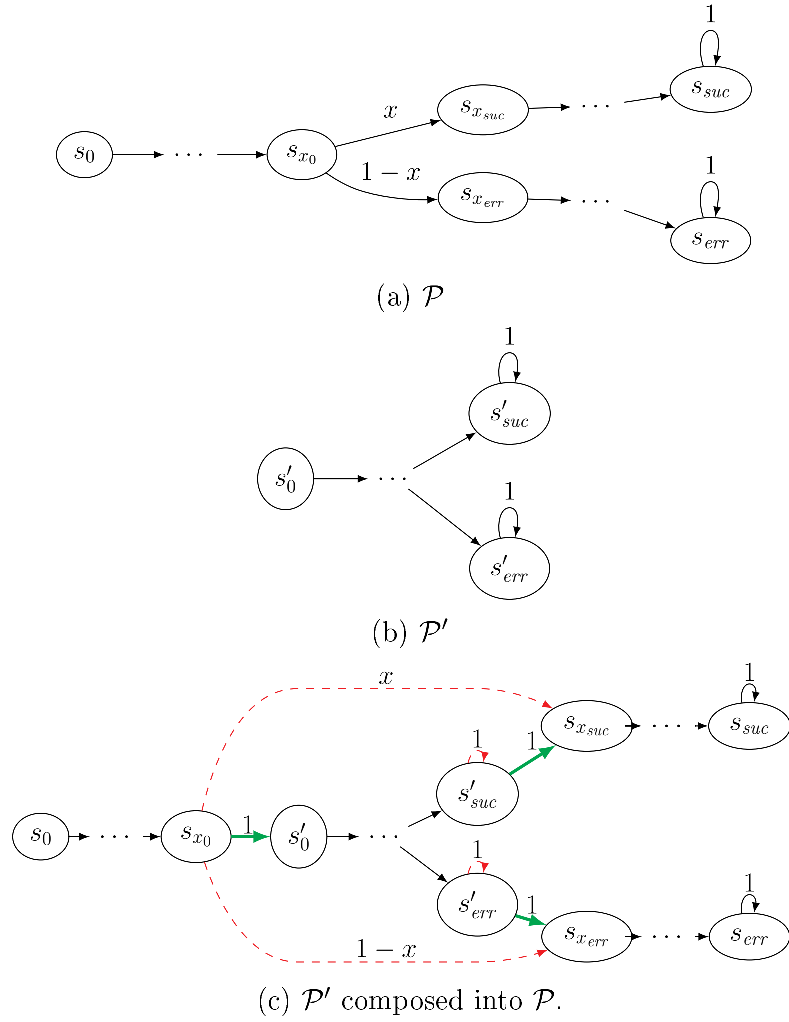
corresponding semantics in an annotative PMC. In a compositional PMC, parameterized transitions relate to the concept of *slots*, whereas annotative PMCs treat variable transitions as behavioral switches. Informally, a slot for the variable x marks the part of a product's behavior where a variant behavior (somehow identified by x) takes place. Note that there can be more than one slot for a given behavior, since a feature may influence behavior at different points of the execution (*behavior scattering*).

With compositional PMCs at hand, we need to be able to derive a DTMC, modeling the behavior of a given product of the product line, as in Section 4.1.1. The intuition is that composition is achieved by connecting the interface of a compositional PMC $\mathcal{P}'$ to the slots in a compositional PMC $\mathcal{P}$ that are meant to abstract the behavior in $\mathcal{P}'$ (see Figure 4.4). In summary, transitions among slot states of $\mathcal{P}$ are removed as well as the looping transitions from success and error absorbing states of $\mathcal{P}'$. Then, slot states are connected to respective interface states, yielding a partially composed PMC. This process is illustrated in Figure 4.4c, which depicts the partial composition of the compositional PMC $\mathcal{P}'$ (Figure 4.4b) into $\mathcal{P}$ (Figure 4.4a) from the perspective of a single slot. New transitions are green bold, while red dashed transitions are the ones suppressed during composition.

A full composition is then obtained by composing PMCs over all slots in a given base compositional PMC at once. After composition, the variability in a compositional PMC is replaced by the variabilities of the PMCs composed into it. In the vending machine (Figure A.2), for instance, if we compose the tea PMC $\mathcal{P}_t$ (Figure 4.3b) into the top-level PMC $\mathcal{P}_\top$ (Figure 4.3a) using the slot $(c_{t_0}, c_{t_{suc}}, c_{t_{err}})$, the resulting compositional PMC will no longer have variable t, but will have a new variable t_{taste}, stemming from $\mathcal{P}_t$. Consequently, to derive a product, one has to recursively perform the composition operation until a plain DTMC is returned.

However, this composition depends upon satisfaction of a presence condition. If the presence condition of a component model is satisfied, this model is composed; otherwise, we compose the *trivial* compositional PMC, instead (Figure 4.5). This compositional PMC models an always successful behavior, so composing it would not affect the overall reliability of the base model.

Figure 4.6 depicts the possible compositions of the tea PMC $\mathcal{P}_t$ (Figure 4.3b) of the vending machine example. If the feature selection contains both features `Tea` and `Taste`, the presence condition $p_{t_{taste}}$ is satisfied, so we compose $\mathcal{P}_{t_{taste}}$ (Figure 4.3c) in the t_{taste} slot (Figure 4.6a). If `Taste` is not selected, $p_{t_{taste}}$ is not satisfied, and we compose the trivial PMC $\widetilde{\mathcal{P}}$ instead (Figure 4.6b).



(a) $\mathcal{P}$

(b) $\mathcal{P}'$

(c) $\mathcal{P}'$ composed into $\mathcal{P}$.

Figure 4.4: Example of a partial composition of PMCs.

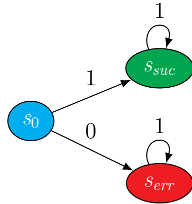
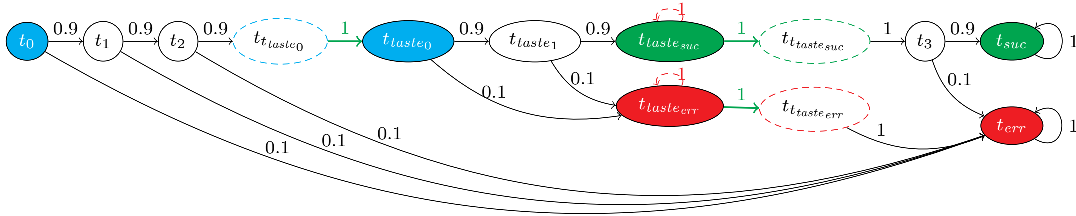


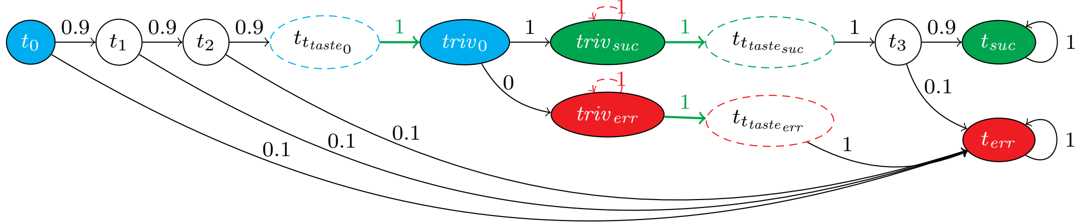
Figure 4.5: Trivial compositional PMC $\widetilde{\mathcal{P}}$.

4.2 Reliability Analysis Strategies

The scenario on which we focus is analyzing the reliability of all products of a product line using model checking of a probabilistic reachability property of Markov-chain models. For this task, one can choose a number of product-line analysis strategies [Thüm et al., 2014]. Following the taxonomy of Thüm et al. [2014], we discussed possible strategies for each of the variability representations (annotative and compositional) presented in Section 4.1.



(a) $\mathcal{P}_{t_{taste}}$ composed into $\mathcal{P}$ (`Taste` is selected).



(b) $\widetilde{\mathcal{P}}$ composed into $\mathcal{P}$ (`Taste` is not selected).

Figure 4.6: Example compositions for the vending machine.

Figure 4.7 depicts these choices. Starting with a compositional (upper left corner) or an annotative model (upper right corner), one can follow any of the outgoing arrows while performing the respective analysis steps (abstracted as functions), until reliabilities are computed (either real-valued reliabilities or an ADD representing all possible values). These analysis steps can be feature-based (green solid arrows), product-based (blue dotted arrows), or family-based (red dashed arrows). Thus, the arrows form an "analysis path" (a function composition), which defines the employed analysis strategy. Furthermore, this diagram is a commuting diagram, as suggested by empirical evidence (Section 3.3.2) and demonstrated later in this section (right-hand side only). This means that different analysis paths are equivalent if they share the start and end points.

After choosing a variability representation, the analysis of any of the resulting models presents another choice: either variability-free models (i.e., DTMC) are derived for each configuration (function λ) and then analyzed (function α), or variability-aware analysis is applied, using some form of parametric model checking (function α_v). The first choice yields a product-based strategy (Section 4.2.1), whereby each variant is independently analyzed. The second one leverages parametric model checking to produce expressions denoting the reliability of PMCs in terms of their variables (Section 2.2.2). These variables carry the semantics they had in the model-checked PMC, so we correspondingly classify the resulting expressions as annotative or compositional.

Evaluating these expressions provides another choice: to evaluate the expressions for each valid configuration (function σ), yielding feature-product-based and family-product-

based (Section 4.2.2.1) strategies; or to interpret the expressions in terms of ADDs (function *lift*), effectively evaluating them for the whole family of models at once (function σ_v)—a step we call *expression lifting*. The latter represents feature-family-based and family-based (Section 4.2.2.2) strategies.

As an example of walking through the choices of Figure 4.7, suppose we start with a compositional model (upper-left corner), perform parametric model checking (move down), and then lift the resulting expressions (move down one more step) and evaluate them (move right), reaching a reliability ADD for the family as a whole. The arrows in this path are, respectively, green solid, red dashed, and red dashed, meaning the analysis strategy is feature-family-based.

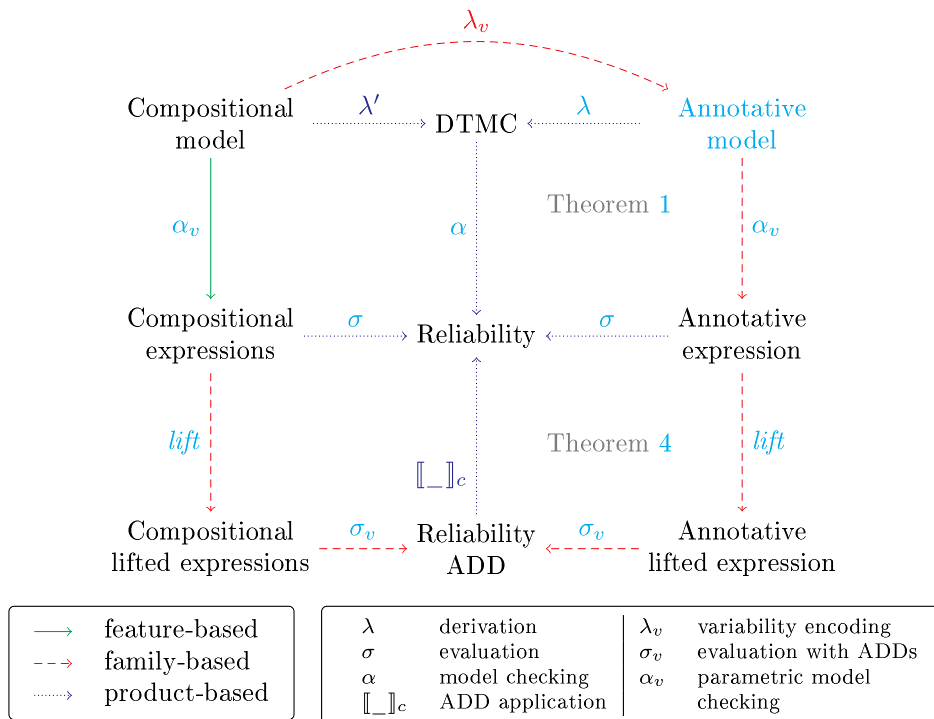


Figure 4.7: Commutative diagram of product-line reliability analysis strategies.

In the remaining sections, we detail some of these strategies and analysis steps with the goal of making statements about commuting relations. Section 4.2.1 presents the product-based analysis strategy for annotative models, with the goal of establishing a baseline for soundness proofs. Section 4.2.2 discusses family-product-based and family-based analyses of annotative models. The formalization of feature-family-based and feature-product-based strategies, as well as variability encoding, is the subject of ongoing research.

4.2.1 Product-based Strategy

Product-based analysis strategies are based on the analysis of generated products or models thereof [Thüm et al., 2014]. In Section 4.1, we have discussed how to represent probabilistic behavioral models of product lines as PMCs, using the annotative approach. There, we also described how to derive models of individual products for the annotative approach. The generated models are plain DTMCs, that is, their variability has been resolved at derivation time. Thus, to analyze the generated models, one only needs to model-check the non-parametric probabilistic reachability for every such model. We hereafter denote this non-parametric model checking analysis step by the following function α.

Definition 9 (Non-parametric model checking)**.** The non-parametric model checking step $\alpha : DTMC \to [0, 1]$ consists of applying the algorithm by Hahn et al. [2011]. For a DTMC $\mathcal{D} = (S, s_0, \mathbf{P}, T)$,

$$\alpha(\mathcal{D}) = Pr^{\mathcal{D}}(s_0, T)$$

Since a DTMC has no parameters, α yields constant functions, which we interpret as plain Real numbers.

Although there are more efficient algorithms for reliability model checking of regular (non-parametric) DTMCs, we use the algorithm by Hahn et al. [2011] in the above definition for uniformity, which eases understanding. Since this algorithm is sound (Lemma 1), a working implementation of the presented theory is free to exploit another sound probabilistic reachability algorithm for performance reasons.

Now we are able to define product-based analysis for annotative models.

Definition 10 (Product-based analysis of annotative models)**.** Given an annotative model $(\mathcal{P}, p, w, FM)$, a product-based analysis yields, for all $c \in [\![FM]\!]$,

$$\alpha(\lambda(\mathcal{P}, w, c))$$

or, alternatively,

$$\alpha([\![\mathcal{P}]\!]_c^w)$$

So, a product-based analysis results in a mapping from configurations to respective reliability values, such as $\{c \mapsto \alpha(\lambda(\mathcal{P}, w, c)) \mid c \in [\![FM]\!]\}$ for annotative models, for instance.

The analysis strategy presented in this section derives models for individual products of a given product line and then apply a single-product analysis technique as is. Since single-product analyses represent the base case upon which product-line analyses are built, the product-based strategy establishes a baseline for proving the soundness of other strategies.

4.2.2 Family-based Strategies

According to Thüm et al. [2014], a family-based analysis strategy is one that (a) operates only on domain artifacts and that (b) incorporates the knowledge about valid feature combinations. In this section, we explore this kind of strategy in the context of annotative probabilistic models, because they encode the behavior of all products of a product line in a single PMC. It is also possible to perform family-based analyses on a compositional model by first transforming it into an annotative one, but the formalization of this variability encoding approach is the subject of ongoing research.

First, we show how to perform an analysis that yields a reliability expression, which can in turn be evaluated for each valid configuration of the product line. This characterizes a family-product-based strategy (Section 4.2.2.1). Then, the aforementioned analysis is leveraged to build a pure family-based (i.e., non-enumerative) strategy (Section 4.2.2.2). At first, it may seem counterintuitive to present the family-product-based approach before the family-based one. However, we shall see that our pure family-based approach builds upon concepts of the hybrid family-product-based approach, and that performing one or the other is a matter of choosing product-based or family-based analysis steps after a preliminary family-based step.

4.2.2.1 Family-product-based Strategy

A family-product-based strategy is a family-based strategy followed by a product-based strategy over intermediate results [Thüm et al., 2014]. The preliminary family-based step of our family-product-based analysis consists of applying parametric model checking of probabilistic reachability (Section 2.2.2) of the underlying PMC of the annotative model. This step is abstracted as a function α_v, where the subscript v denotes that it is a variability-aware version of the non-parametric model checking function α (Definition 9).

Definition 11 (Parametric model checking)**.** The parametric model checking analysis step $\alpha_v : PMC_X \to \mathcal{F}_X$ consists of applying the algorithm by Hahn et al. [2011] for probabilistic reachability, which yields a rational expression $\varepsilon \in \mathcal{F}_X$ for a PMC with

variables set X. For a PMC $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$, the input target states of the algorithm are the ones in T.

After performing parametric model checking, the result of reachability analysis is an expression over the same variables as the annotative input PMC, denoting the PMC's reliability as a function of these variables. Hence, we expect this annotative reliability expression to be evaluated using the same evaluation functions that restricted the possible behaviors in the original model. This *expression evaluation*, which can be seen as model derivation applied to expressions, is captured in function σ.

Definition 12 (Expression evaluation)**.** Given an expression ε over a set X of variables, an evaluation factory w, and a configuration $c \in [\![FM]\!]$, we define the expression evaluation function in a similar fashion as DTMC derivation:

$$\sigma(\varepsilon, w, c) = \varepsilon[X/w(c)]$$

Likewise, we can use $[\![\varepsilon]\!]_c^w$ to denote $\sigma(\varepsilon, w, c)$.

The function σ is applied to the reliability expression for all valid configurations of the product line, yielding the final product-based step. The resulting family-product-based approach for the analysis of annotative models is then defined as follows.

Definition 13 (Family-product-based analysis)**.** Given an annotative model $(\mathcal{P}, p, w, FM)$, the family-product-based analysis yields, for all $c \in [\![FM]\!]$,

$$\sigma(\alpha_v(\mathcal{P}), w, c)$$

or, alternatively,

$$[\![\alpha_v(\mathcal{P})]\!]_c^w$$

Figure 4.8 illustrates the family-product-based strategy in contrast with the product-based one (Section 4.2.1), providing an intuition for why they commute. DTMC derivation λ and expression evaluation σ are both performed for a configuration c such that $c \models p_x$. This way, $w(c)(x) = 1$ and the reliability is 0.9801. If c were such that x was absent (i.e., $c \not\models p_x$), then the reliability would be 0.99.

To be considered sound, a family-product-based analysis must be equivalent to performing a product-based analysis of all products. This means that performing a parametric model checking step and then evaluating the resulting expression for each valid product must yield the same result as first deriving the original annotative model for each product and then performing non-parametric model checking on each resulting DTMC. To prove that this equivalence holds, we can leverage a more general result about PMCs and well-defined evaluations.

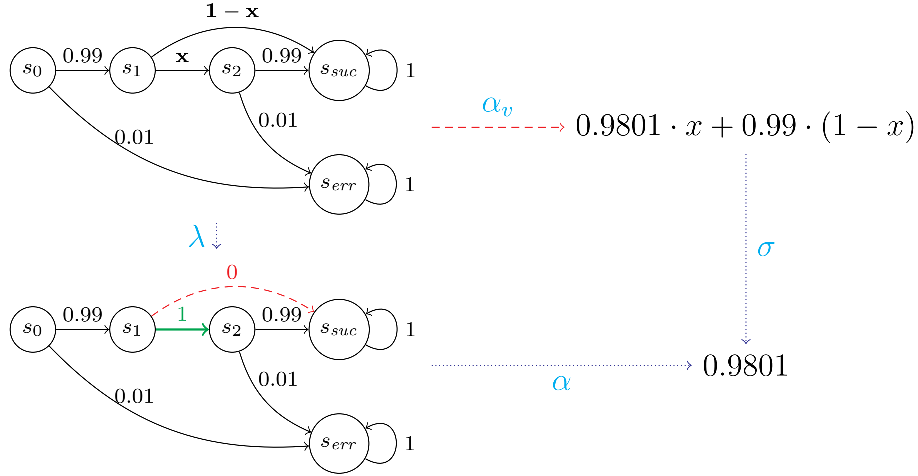


Figure 4.8: Example of family-product-based analysis (α_v followed by σ) in contrast to a product-based analysis (λ followed by α) of an annotative PMC.

Lemma 3 (Commutativity of PMC and expression evaluations)**.** *Given any PMC* $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$ *and a well-defined evaluation* u*, it holds that*

$$\alpha(\mathcal{P}[X/u]) = \alpha_v(\mathcal{P})[X/u]$$

Proof.

$$\begin{aligned} \alpha(\mathcal{P}[X/u]) &= \alpha(\mathcal{P}_u) && \text{(syntax change)} \\ &= Pr^{\mathcal{P}_u}(s_0, T) && \text{(Definition 9)} \end{aligned}$$

and, since u is well-defined,

$$= \alpha_v(\mathcal{P})[X/u] \qquad \text{(Lemma 1 and Definition 11)}$$

□

Using this result, we are able to express the soundness of the family-product-based approach in the following theorem.

Theorem 1 (Soundness of family-product-based analysis)**.** *Given an annotative model* $(\mathcal{P}, p, w, FM)$*, for all* $c \in [\![FM]\!]$

$$\alpha([\![\mathcal{P}]\!]_c^w) = [\![\alpha_v(\mathcal{P})]\!]_c^w$$

Alternatively, $\alpha(\lambda(\mathcal{P}, w, c)) = \sigma(\alpha_v(\mathcal{P}), w, c)$*.*

Proof. Since $w(c)$ is a well-defined evaluation (Lemma 2), we can use it to instantiate u in Lemma 3. Thus, let $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$.

$$\begin{aligned} \alpha(\llbracket \mathcal{P} \rrbracket_c^w) &= \alpha(\mathcal{P}[X/w(c)]) && \text{(Definition 8)} \\ &= \alpha_v(\mathcal{P})[X/w(c)] && \text{(Lemmas 2 and 3)} \\ &= \llbracket \alpha_v(\mathcal{P}) \rrbracket_c^w && \text{(Definition 12)} \end{aligned}$$

□

As a major result, Theorem 1 states that the diagram in Figure 4.9 commutes. This diagram corresponds to the upper right quadrant in Figure 4.7.

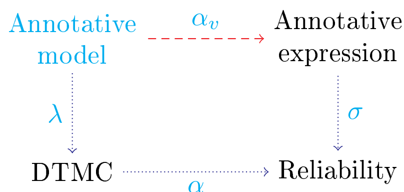


Figure 4.9: Statement of Theorem 1.

4.2.2.2 Family-based Strategy

The pure family-based strategy starts by applying parametric model checking to the given annotative model, as in the family-based step of the family-product-based strategy. However, instead of evaluating the resulting expression for each variant, we *lift* it to an ADD-based expression, which can be evaluated for all variants at once. While an expression is evaluated with real values, a lifted expression is evaluated using ADDs that represent Boolean functions from features to real values. Each of these ADDs encode the values that a variable can assume according to each possible configuration, also known as *variational data* [Walkingshaw et al., 2014]. Since this approach incorporates the knowledge of valid feature combinations, it is a family-based strategy.

Let us take the vending machine product line (Figure A.1) as an example. Its reliability expression after parametric model checking has 8 terms, one of which is $0.124659 \cdot t \cdot t_{taste}$. Starting from the evaluation factory w, we can derive functions ψ_x that, for each variable x, take a configuration $c \in \llbracket FM \rrbracket$ as input and output the corresponding value $w(c)(x)$.

For t and t_{taste}, for instance, these functions would be as follows:

$$\begin{aligned}
\psi_t(\texttt{Tea}, \neg\texttt{Soda}, \neg\texttt{Taste}) &= 1 & \qquad \psi_{t_{taste}}(\texttt{Tea}, \neg\texttt{Soda}, \neg\texttt{Taste}) &= 0 \\
\psi_t(\texttt{Tea}, \neg\texttt{Soda}, \texttt{Taste}) &= 1 & \psi_{t_{taste}}(\texttt{Tea}, \neg\texttt{Soda}, \texttt{Taste}) &= 1 \\
\psi_t(\neg\texttt{Tea}, \texttt{Soda}, \neg\texttt{Taste}) &= 0 & \psi_{t_{taste}}(\neg\texttt{Tea}, \texttt{Soda}, \neg\texttt{Taste}) &= 0 \\
\psi_t(\neg\texttt{Tea}, \texttt{Soda}, \texttt{Taste}) &= 0 & \psi_{t_{taste}}(\neg\texttt{Tea}, \texttt{Soda}, \texttt{Taste}) &= 0
\end{aligned}$$

Having each of these functions represented by an ADD enables the efficient computation of the reliability expression as another ADD $\hat{r}$, representing a Boolean function that could be defined pointwise as $\hat{r}(c) = 0.124659 \cdot \psi_t(c) \cdot \psi_{t_{taste}}(c)$ (we omit the remaining terms for simplicity).

We now formally define expression lifting, as well as the mechanics of generating ADD-based evaluations and evaluating lifted expressions.

Definition 14 (Expression lifting)**.** For a given rational expression $\varepsilon \in \mathcal{F}_X$, whose semantics is a rational function $\mathbb{R}^{|X|} \to \mathbb{R}$, and a product line with k features, we define the lifted expression $\mathit{lift}(\varepsilon) = \hat{\varepsilon}$ as an expression which is syntactically equal to ε, but whose semantics is lifted to a rational function $(\mathbb{B}^k \to \mathbb{R})^{|X|} \to (\mathbb{B}^k \to \mathbb{R})$, such that:

- The function's inputs are k-ary ADDs.
- Polynomial coefficients are interpreted as constant ADDs (e.g., the number 5 becomes $c \in \mathbb{B}^k \mapsto 5$). We denote a constant a lifted to a constant ADD as $\hat{a}$, so that $\hat{a}(\bar{b}) = a$ (where $\bar{b}$ is a Boolean tuple).
- Arithmetic operators are lifted to their ADD-based counterparts.

Hence, the admitted evaluations for $\hat{\varepsilon}$ are of type $u : X \to (\mathbb{B}^k \to \mathbb{R})$, so that variables are properly replaced by k-ary ADDs.

By the above definition, lifted expressions are syntactically equal to their original (non-lifted) counterparts. However, instead of using Real arithmetics, we interpret operators, constants, and variables using ADDs and ADD arithmetics (Section 2.3). These semantically lifted expressions are sound in the sense that they denote functions that, when evaluated with a given configuration, yield the same results as if the variables of the original expressions would have been individually evaluated for the same configuration.

Lemma 4 (Soundness of expression lifting)**.** *If ε is a rational expression over Real constants and variables $x_i \in X$, $|X| = n$, $A_1, \ldots, A_n$ are ADDs, and $\hat{\varepsilon} = \mathit{lift}(\varepsilon)$, then*

$$\hat{\varepsilon}[x_1/A_1, \ldots, x_n/A_n](\bar{b}) = \varepsilon[x_1/A_1(\bar{b}), \ldots, x_n/A_n(\bar{b})]$$

where $\bar{b}$ is a vector of k Booleans, corresponding to a selection of the k features in a given product line.

Proof. The proof is by structural induction on expression ε. The base cases are constant expressions and single variables:

- $\varepsilon = c$, where $c \in \mathbb{R}$:

  In this case, $\hat{\varepsilon} = \hat{c}$. Since ε has no variables (and neither has $\hat{\varepsilon}$), we apply the empty evaluation $[\,]$. Thus, $\hat{\varepsilon}[\,](\bar{b}) = \hat{c}(\bar{b}) = c = \varepsilon = \varepsilon[\,]$.

- $\varepsilon = x$:

  In this case, $\hat{\varepsilon} = x$. If A is an arbitrary ADD, then: $\hat{\varepsilon}[x/A](\bar{b}) = A(\bar{b}) = \varepsilon[x/A(\bar{b})]$.

As induction hypothesis, for the expressions $\varepsilon = \varepsilon_1$ and $\varepsilon = \varepsilon_2$, assume that the following holds:

$$\hat{\varepsilon}[x_1/A_1, \ldots, x_n/A_n](\bar{b}) = \varepsilon[x_1/A_1(\bar{b}), \ldots, x_n/A_n(\bar{b})] \tag{I.H.}$$

Let $u : X \to (\mathbb{B}^k \to \mathbb{R})$ be a lifted evaluation such that $u(x_i) = A_i$ is an ADD. Since ε is a rational expression (i.e., a quotient of polynomials, as yielded by a parametric model checking algorithm—see Sections 2.2.1 and 2.2.2), it involves only the four basic arithmetic operators and exponentiation to Natural powers. Thus, we examine the cases where we perform ADD arithmetics (Section 2.3), corresponding to the allowed operations in ε:

- $\varepsilon = \varepsilon_1 \odot \varepsilon_2$, where $\odot \in \{+, -, \times, \div\}$:

  In this case, $\hat{\varepsilon} = \hat{\varepsilon_1} \odot \hat{\varepsilon_2}$. Hence,

$$\begin{aligned}
\hat{\varepsilon}[X/u](\bar{b}) &= \big(\hat{\varepsilon_1} \odot \hat{\varepsilon_2}\big)[X/u](\bar{b}) \\
&= \big(\hat{\varepsilon_1}[X/u] \odot \hat{\varepsilon_2}[X/u]\big)(\bar{b}) && \text{(evaluation)} \\
&= \hat{\varepsilon_1}[X/u](\bar{b}) \odot \hat{\varepsilon_2}[X/u](\bar{b}) && \text{(ADD arithmetics)} \\
&= \hat{\varepsilon_1}[x_1/A_1, \ldots, x_n/A_n](\bar{b}) \\
&\quad \odot \hat{\varepsilon_2}[x_1/A_1, \ldots, x_n/A_n](\bar{b}) && \text{(expanding } u\text{)} \\
&= \varepsilon_1[x_1/A_1(\bar{b}), \ldots, x_n/A_n(\bar{b})] \\
&\quad \odot \varepsilon_2[x_1/A_1(\bar{b}), \ldots, x_n/A_n(\bar{b})] && \text{(induction hypothesis)} \\
&= \big(\varepsilon_1 \odot \varepsilon_2\big)[x_1/A_1(\bar{b}), \ldots, x_n/A_n(\bar{b})] && \text{(evaluation)} \\
&= \varepsilon[x_1/A_1(\bar{b}), \ldots, x_n/A_n(\bar{b})]
\end{aligned}$$

- $\varepsilon = \varepsilon_1^i$, where $i \in \mathbb{N}$:

In this case, $\hat{\varepsilon} = \hat{\varepsilon_1}^i$. Hence,

$$
\begin{aligned}
\hat{\varepsilon}[X/u](\bar{b}) &= \hat{\varepsilon_1}^i[X/u](\bar{b}) \\
&= \hat{\varepsilon_1}[X/u]^i(\bar{b}) && \text{(evaluation)} \\
&= \hat{\varepsilon_1}[X/u](\bar{b})^i && \text{(ADD arithmetics)} \\
&= \hat{\varepsilon_1}[x_1/A_1, \ldots, x_n/A_n](\bar{b})^i && \text{(expanding } u\text{)} \\
&= \varepsilon_1[x_1/A_1(\bar{b}), \ldots, x_n/A_n(\bar{b})]^i && \text{(induction hypothesis)} \\
&= \varepsilon_1^i[x_1/A_1(\bar{b}), \ldots, x_n/A_n(\bar{b})] && \text{(evaluation)} \\
&= \varepsilon[x_1/A_1(\bar{b}), \ldots, x_n/A_n(\bar{b})]
\end{aligned}
$$

□

Note how a lifted expression demands a different type of evaluation, namely one that replaces variables with ADDs. To handle this interdependency, we correspondingly lift the evaluation factory.

Definition 15 (Lifted evaluation factory)**.** Given an evaluation factory w defined over a feature model FM and a set X of variables, the factory's lifted counterpart is a function $\hat{w} : X \to (\mathbb{B}^{|FM|} \to \mathbb{R})$ that yields an ADD for a given variable. This function is such that, for every variable $x \in X$ and all $c \in [\![FM]\!]$,

$$\hat{w}(x)(c) = w(c)(x)$$

With a lifted evaluation factory, one can evaluate a lifted expression over the same set X in a variability-aware fashion. The intuition is that we valuate each variable with an ADD that encodes all the real values it may assume for any configuration of the product line.

Definition 16 (Variability-aware expression evaluation)**.** Let $\hat{w}$ be a lifted evaluation factory and $\hat{\varepsilon}$ be a lifted expression. The variability-aware expression evaluation function, σ_v, is defined as

$$\sigma_v(\hat{\varepsilon}, \hat{w}) = \hat{\varepsilon}[X/\hat{w}]$$

Remark 2. This definition of variability-aware evaluation is not restricted to reliability analysis or to the specific definitions of probabilistic models presented in this text. Indeed, one can notice that it relies on the definitions of an expression with rational function semantics and of an evaluation factory with respect to a given feature model.

Thus, we are able to prove the following theorem, which applies to product line analysis strategies that are based on expression evaluation.

Theorem 2 (Soundness of variability-aware expression evaluation). *If ε is an expression and w is an evaluation factory with respect to a feature model FM, let $\hat{\varepsilon}$ and $\hat{w}$ be their respective lifted counterparts. Then, for all $c \in [\![FM]\!]$,*

$$\sigma_v(\hat{\varepsilon}, \hat{w})(c) = \sigma(\varepsilon, w, c)$$

In other words, $\hat{\varepsilon}[X/\hat{w}](c) = \varepsilon[X/w(c)]$.

Proof. Using $\hat{w}$ as a substitution,

$$\hat{\varepsilon}[X/\hat{w}] = \hat{\varepsilon}[x_1/\hat{w}(x_1), \ldots, x_n/\hat{w}(x_n)]$$

Thus, for all $c \in [\![FM]\!]$,

$$\begin{aligned}
\sigma_v(\hat{\varepsilon}, \hat{w})(c) &= \hat{\varepsilon}[X/\hat{w}](c) && \text{(Definition 16)} \\
&= \hat{\varepsilon}[x_1/\hat{w}(x_1), \ldots, x_n/\hat{w}(x_n)](c) \\
&= \varepsilon[x_1/\hat{w}(x_1)(c), \ldots, x_n/\hat{w}(x_n)(c)] && \text{(Lemma 4)} \\
&= \varepsilon[x_1/w(c)(x_1), \ldots, x_n/w(c)(x_n)] && \text{(Definition 15)} \\
&= \varepsilon[X/w(c)] \\
&= \sigma(\varepsilon, w, c) && \text{(Definition 12)}
\end{aligned}$$

□

We have seen that, in a product line with feature model FM, the presence function p denotes a presence condition p_x as a Boolean function $p(x) : [\![FM]\!] \to \mathbb{B}$. Since this can be alternatively expressed as $p(x) : \mathbb{B}^{|FM|} \to \mathbb{B}$, the presence function can also be encoded by ADDs, denoted by $\hat{p}(x)$. We now resort to the pointwise definition of w as $w(c)(x) = p(x)(c)$ (Remark 1), to define a lifted evaluation factory $\hat{w}$, for evaluating the lifted version of expressions resulting from parametric model checking of an annotative model.

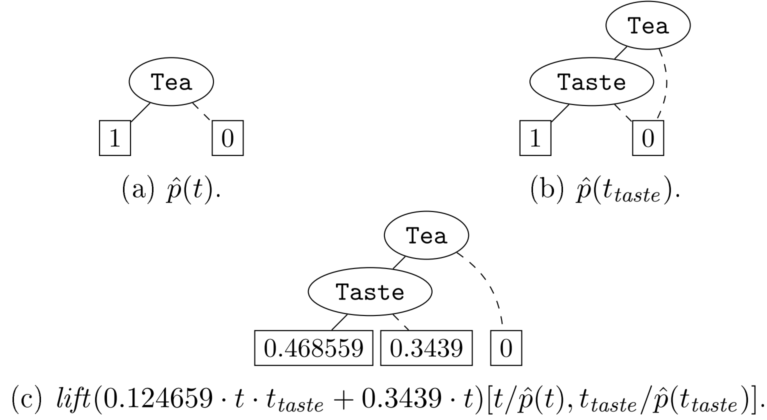
Lemma 5 (Soundness of lifted annotative evaluation factory). *Given an annotative model $(\mathcal{P}, p, w, FM)$ and a function $\hat{p} : X \to (\mathbb{B}^{|FM|} \to \mathbb{B})$ that encodes presence conditions for variables as ADDs, then $\hat{w} = \hat{p}$ is a lifted evaluation factory for w.*

Proof. From Definition 7, we have that

$$w(c)(x) = \begin{cases} 1 & \text{if } p(x)(c) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Thus, from Remark 1, $w(c)(x) = p(x)(c)$. Also, $p(x)(c) = \hat{p}(x)(c)$ by definition, so $w(c)(x) = \hat{p}(x)(c)$. □

Recalling the vending machine example, the presence conditions for the variables t and t_{taste} are, respectively, `Tea` and `Tea` $\wedge$ `Taste`. Then, the ADDs $\hat{p}(t)$ and $\hat{p}(t_{taste})$ are given by the Figures 4.10a and 4.10b, where we use the notation presented in Section 2.3. If we evaluate a lifted version of the example expression $\varepsilon = 0.124659 \cdot t \cdot t_{taste} + 0.3439 \cdot t$ (2 terms from the actual reliability expression for the vending machine annotative model in Figure A.1) with $\hat{p}$, the resulting ADD will be $\hat{r} = 0.124659 \cdot \hat{p}(t) \cdot \hat{p}(t_{taste}) + 0.3439 \cdot \hat{p}(t)$, as depicted in Figure 4.10c. Hence, for a given configuration $c \in [\![FM]\!]$, if both `Tea` and `Taste` are present (i.e., $\hat{p}(t)(c) = 1$ and $\hat{p}(t_{taste})(c) = 1$), then $\hat{r}(c) = 0.124659 \cdot 1 \cdot 1 + 0.3439 \cdot 1 = 0.468559$; if only `Tea` is present, then $\hat{r}(c) = 0.124659 \cdot 1 \cdot 0 + 0.3439 \cdot 1 = 0.3439$; and if both `Tea` and `Taste` are absent, then $\hat{r}(c) = 0$.



(a) $\hat{p}(t)$.

(b) $\hat{p}(t_{taste})$.

(c) $\mathit{lift}(0.124659 \cdot t \cdot t_{taste} + 0.3439 \cdot t)[t/\hat{p}(t), t_{taste}/\hat{p}(t_{taste})]$.

Figure 4.10: Example of lifted expression evaluation using $\hat{p}$.

Using the result from Lemma 5, we can now express the soundness of this family-based analysis step of evaluating lifted expressions.

Theorem 3 (Soundness of expression evaluation using $\hat{p}$). *Given an annotative model $(\mathcal{P}, p, w, FM)$, $\varepsilon = \alpha_v(\mathcal{P})$, and $\hat{\varepsilon} = \mathit{lift}(\varepsilon)$, let $\hat{p}$ be the encoding of the presence condition function p to yield ADDs. If we use $\hat{p}$ as a lifted evaluation factory, then for all $c \in [\![FM]\!]$*

$$[\![\sigma_v(\hat{\varepsilon}, \hat{p})]\!]_c = [\![\varepsilon]\!]_c^w$$

Alternatively, $\sigma_v(\mathit{lift}(\varepsilon), \hat{p})(c) = \sigma(\varepsilon, w, c)$.

Proof. For a given annotative model, Lemma 5 states that $\hat{p}$ is a sound lifted counterpart of w. Hence, by Theorem 2, $\varepsilon[X/w(c)] = \hat{\varepsilon}[X/\hat{p}](c)$. In other words, $[\![\sigma_v(\hat{\varepsilon}, \hat{p})]\!]_c = [\![\varepsilon]\!]_c^w$. □

Figure 4.11 illustrates the main result from Theorem 3. The depicted diagram, which corresponds to the lower right quadrant in Figure 4.7, is commutative because of this theorem.

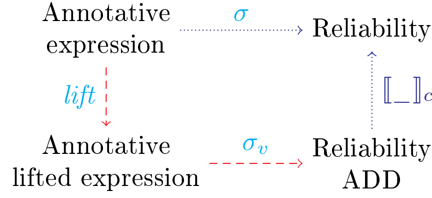


Figure 4.11: Statement of Theorem 3.

Now that we have all analysis steps needed, we can formally define the family-based strategy.

Definition 17 (Family-based analysis)**.** Given an annotative model $(\mathcal{P}, p, w, FM)$, a family-based analysis yields, for all $c \in [\![FM]\!]$,

$$\sigma_v\big(\mathit{lift}(\alpha_v(\mathcal{P})), \hat{p}\big)(c)$$

or, alternatively,

$$[\![\sigma_v\big(\mathit{lift}(\alpha_v(\mathcal{P})), \hat{p}\big)]\!]_c$$

This definition may seem also enumerative at first, but the result is, in fact, a Boolean function (encoded as an ADD). The function application to a given configuration is meant only as a comparison to the other strategies. Indeed, family-based analysis is sound if, and only if, it yields an ADD for which every valid configuration $c \in [\![FM]\!]$ results in the same probability as if the original annotative model had been subject to product-based analysis for the same configuration c.

Theorem 4 (Soundness of family-based analysis)**.** *Given an annotative model* $(\mathcal{P}, p, w, FM)$*, for all* $c \in [\![FM]\!]$ *it holds that*

$$[\![\sigma_v\big(\mathit{lift}(\alpha_v(\mathcal{P})), \hat{p}\big)]\!]_c = \alpha([\![\mathcal{P}]\!]_c^w)$$

Proof. Follows from the successive application of Theorems 3 and 1:

$$\begin{aligned} [\![\sigma_v\big(\mathit{lift}(\alpha_v(\mathcal{P})), \hat{p}\big)]\!]_c &= [\![\alpha_v(\mathcal{P})]\!]_c^w && \text{(Theorem 3)} \\ &= \alpha([\![\mathcal{P}]\!]_c^w) && \text{(Theorem 1)} \end{aligned}$$

□

As a key result, Theorem 4 states that the diagram in Figure 4.12 commutes. This diagram corresponds to the right half of the one in Figure 4.7.

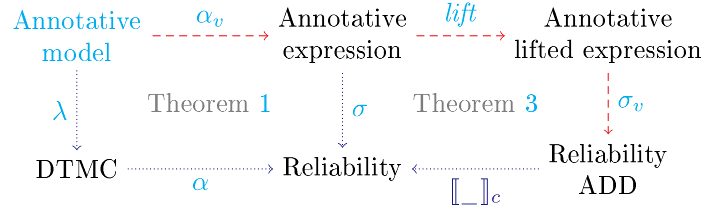


Figure 4.12: Statement of Theorem 4.

Together, the theorems demonstrated in this section constitute the main contribution of this work. Intermediate steps of the presented analysis techniques commute, making the annotation-based part (right-hand side) of the diagram in Figure 4.7 fully commutative[1]. Thus, any path constructed by following the arrows in that part of the diagram yields an analysis that is equivalent to the one yielded by any other path that shares the same starting and ending points. This way, we guarantee that the product-based, family-based, and family-product-based reliability analysis techniques presented in this work yield the same results if given the same input models. Furthermore, we formally described the different analysis strategies in terms of reusable functions, making them comparable to one another. Equivalent specification and proofs for the feature-based part (left-hand side) of the diagram in Figure 4.7 are an ongoing effort.

[1] Indeed, the right side of the diagram in Figure 4.7 corresponds to a transposed version of Figure 4.12.

Chapter 5

Conclusion

We formally presented three approaches to reliability analysis of product lines, covering the product-based, family-based, and family-product-based strategies in the taxonomy by Thüm et al. [2014]. The soundness of our analysis techniques is established by results on the commutativity of their intermediate steps, summarized by the commuting diagram in Figure 4.7. This constitutes formal evidence that, given a product line, each of the formalized approaches yields the same results as the others, enabling practitioners to choose among analysis strategies based on their space and time trade-offs[1].

The input models for our analysis approaches are based on the formalism of parametric Markov chains, meaning they can be represented using the input language of parametric model checkers such as PRISM [Kwiatkowska et al., 2011] and PARAM [Hahn et al., 2010]. Indeed, the parametric probabilistic reachability algorithm by Hahn et al. [2011], used throughout this work as an instance of variability-aware analysis function (α_v), is implemented by these tools. We have implemented most of the presented strategies as a product line of product-line analysis tools, using PARAM as the parametric model checker. Our product line, which is publicly available as free and open-source software at `https://github.com/SPLMC/reana-spl`, served as the basis for the development of our theory. Moreover, to the best of our knowledge, this is the first model checking tool to implement all three dimensions of analysis.

We empirically compared the results obtained by the different analysis strategies implemented in our tool, improving our confidence in our theoretical findings that they are, indeed, commutative. This empirical comparison also provided evidence of the commutativity of the strategies that we have not yet formalized. Moreover, we had our proofs reviewed by fellow researchers outside our group, and the resulting theory is being prepared for submission to a peer-reviewed journal. To reduce the risk of a mismatch

[1] These trade-offs are not presented in this work, as they still need to be identified.

between the implementation and the derived theory, we implemented our product line, ReAna-SPL, using functional programming principles and techniques.

We also employed graphs to map dependencies between elements in our theory. Such graphs assisted in the task of maintaining consistency and correctness through theory refactorings. For instance, at some point a reviewer identified inconsistencies between Definitions 8, 12 and 16. By locating these definitions in Figure B.1 and following the arrows in reverse order, we were able to track directly and transitively dependent elements that potentially needed to adjust to a change in notation.

Although our theory is focused on reliability analysis, we were able to prove a general result on lifting rational functions over the Real numbers to work with ADDs (Lemma 4). This result can be leveraged to evaluate algebraic expressions in the context of product lines.

5.1 Related Work

Efficient analysis of software product lines is a relevant problem that has been tackled from many different perspectives, as pointed out by a recent survey [Thüm et al., 2014]. In particular, several model checking techniques have been successfully lifted to work with product lines [Apel et al., 2013b; Chrszon et al., 2016; Classen et al., 2013, 2011, 2014; Dubslaff et al., 2015; Ghezzi and Molzam Sharifloo, 2013; Kowal et al., 2015; Nunes et al., 2012]. In contrast to existing research in this area, our work presents different analysis techniques, covering all but one of the strategies identified in the taxonomy by Thüm et al. [2014] (the exception being the feature-family-product-based strategy). A similarity, on the other hand, is that our formalization effort is also focused on the family-based and product-based dimensions of product-line analysis. In what follows, we discuss the closest related work according to different criteria.

PMC-based analysis of product lines: Ghezzi and Molzam Sharifloo [2013] propose a model-based approach to analyze non-functional properties of product lines, illustrated by reliability and energy-consumption analysis. Their technique models probabilistic behavior by organizing parametric Markov chains in a hierarchical data structure, derived from nested UML sequence diagrams, annotated with the reliability of individual operations. Then, they employ parametric model checking in a bottom-up fashion, yielding a hierarchy of reliability expressions that are evaluated for each product configuration of interest. Although Ghezzi and Molzam Sharifloo also deal with modeling issues, their analysis technique can be seen as an instance of the *feature-product-based* reliability analysis in our framework, where the PMCs obtained from the nested sequence diagrams

form the set of compositional PMCs, and the decomposition tree induces the dependency relation between them.

Rodrigues et al. [2015] introduced *Featured Discrete-Time Markov Chains* (FDTMC), an extension of DTMCs to cope with variability and to represent the probabilistic behavior of product lines. This formalism, which is not restricted to reliability, enables verification of any probabilistic property that can be expressed using *Probabilistic Computation Tree Logic* (PCTL) [Hansson and Jonsson, 1994]. The authors present three *family-based* approaches to conduct such analyses, one of which relies on an encoding of an FDTMC as a PMC to leverage off-the-shelf model checkers. Our work, in contrast, relies on models specifically tailored to reliability analysis (a probabilistic reachability property), but incorporates different strategies to perform this analysis, covering the currently accepted product-line analysis taxonomy [Thüm et al., 2014]. Furthermore, Rodrigues et al. do not formally argue about the soundness of their approaches.

The framework we present can be leveraged to represent FDTMCs, provided that the reliability-specific constraints to PMCs are relaxed. We can say that any PMC $(S, s_0, X, \mathbf{P}, T)$, along with an evaluation factory w and a feature model FM, represents an FDTMC (S, ν, FM, Π) such that, for all $s, s' \in S$ and $c \in [\![FM]\!]$:

- $\Pi(s, s')(c) = \mathbf{P}(s, s')[X/w(c)]$; and
- $\nu(s) = \begin{cases} 1 & \text{if } s = s_0 \\ 0 & \text{otherwise} \end{cases}$

Feature-based model checking: Li et al. [2005] and Liu et al. [2010] have proposed feature-based approaches to the analysis of non-probabilistic temporal properties of product lines. Using models of feature behavior based on transition systems and required properties expressed with Computation Tree Logic (CTL) [Clarke and Emerson, 1982], they analyze each feature in isolation and generate partial results that can be later reused. The composition of features in their proposed models relies on interface states, a concept that we leveraged to define PMC interfaces and slots. However, the interfaces defined by Li et al. [2005] can have an arbitrary number of outgoing states, and Liu et al. [2010] extended them to support inter-feature cycles. Our use of interfaces, in contrast, is focused on reliability analysis (a probabilistic existence property expressed in PCTL), allowing us to define two outgoing states to abstract success and error conditions, while also ruling out the existence of cycles. Moreover, both Li et al. [2005] and Liu et al. [2010] treat feature modules as open systems, so they aggregate partial analysis results and CTL obligations to the interfaces themselves. Since we focus on a compositional model of a single product line, we use a separate model for intermediate feature reliability expressions. Because of these differences in modeling and in the nature of analyzed properties,

we see their work and our own as complementary. Nonetheless, their work presents formal specification and soundness proofs in the feature-based dimension of analysis, whereas we present such formal elements for the family-based dimension.

Family-based model checking: Dubslaff et al. [2015] created a framework for modeling probabilistic and non-deterministic properties of dynamic product lines. This framework consists of modeling the behaviors of features in isolation, yielding models that are later composed into a family-based model. The models and their compositions are established in terms of *Markov Decision Processes* (MDP), enabling their representation in a way that allows the composed model to be model-checked using off-the-shelf tools [Chrszon et al., 2016]. The focus of their work is on modeling probabilistic behavior of product lines in a way that existing model checking techniques can be exploited. In contrast, our goal is to prove soundness of alternative analysis strategies, leaving modeling issues out of scope. Although their modeling and analysis technique is sufficiently general to enable reliability analysis of static product lines, which are our focus, it enables only family-based and product-based strategies (which the authors call, respectively, *all-in-one* and *one-by-one* [Dubslaff et al., 2015]), whereas our work also includes the feature-based dimension. Nonetheless, their family-based technique is an alternative to ours, since it encodes the feature model's constraints in the behavioral model itself.

Kowal et al. [2014] presented a formalism to describe performance models of product lines in a compositional fashion, based on performance-annotated activity diagrams described in a delta-oriented language. Similar to our work, they provide formal definitions and provide theorems stating the soundness of their approach (although proofs are not provided in the paper). On the other hand, the semantics of their diagrams is expressed by continuous-time Markov chains (CTMC), which are more appropriate to performance analysis than DTMCs. Because of that, the two pieces of work complement each other. Future work could investigate the feasibility of defining alternative analysis strategies using their models and an approach similar to ours.

Variability encoding: Previous research has exploited variability encoding (also called configuration lifting) as a technique to produce family-based model checking of product lines [Apel et al., 2011, 2013b; Kowal et al., 2015; Post and Sinz, 2008]. von Rhein et al. [2016] formalize variability encoding in the context of programming languages, that is, the transformation of compile-time variability into load-time variability. This transformation is realized using *if-then-else* operations and an encoding of features as control variables in the resulting program, which the authors call a variant simulator. They prove their transformation preserves the behavior of variants in the variability-encoded program for corresponding configurations. The concept of encoding variability in a simulator, as mentioned before, inspired our notion of variability encoding for PMCs.

However, whereas von Rhein et al. [2016] prove the soundness of their variability encoding approach (using trace semantics and a weak bisimulation relation to correlate behaviors), we do not provide a formalization of our corresponding approach.

Formal approaches to variability-aware analysis: The definition of product-line analysis techniques that are sound by construction has been investigated recently [Bodden et al., 2013; Chen and Erwig, 2014; Midtgaard et al., 2015], although not specifically in the context of model checking. Midtgaard et al. [2015] presented a methodology to derive family-based static analyses from single-product analyses based on *abstract interpretation*. This approach enables the lifting of existing analyses to work with product lines, yielding variability-aware analyses that are correct by construction. Although the authors only walked through a data-flow analysis scenario, they claim the methodology could be applied to other analyses, including model checking. Similar to their work, we provide soundness proofs of product-line analyses, conditioned on the soundness of a given single-product analysis. However, we do not provide a framework for derivation of analysis strategies in general; instead, we focus on providing formal evidence that a set of alternative strategies for reliability analysis are sound, while also highlighting the relations between their intermediate steps. In this sense, our work can also be seen as a preliminary investigation on deriving alternative strategies to perform a given analysis.

Comparison of analysis dimensions: Kolesnikov et al. [2014] empirically compared family-based, feature-based, and product-based type checking of Java-based product lines. Their work was the first empirical study covering all three dimensions of analysis, providing guidance to practitioners over which type checking strategy to apply for a given product line. In a sense, their research and our own are complementary, since each one deals with a different analysis type (type checking and model checking). However, in contrast with their work, our focus is on the formal aspects of analysis—although we argue our tool can be leveraged to perform empirical studies in future work. Furthermore, Kolesnikov et al. neither investigate combined strategies nor prove the soundness of the implemented type checkers.

von Rhein et al. [2013] proposed a model for classification and comparison of product-line analyses (the *PLA model*), whereby existing analyses are broken down into intermediate steps. This model abstracts possible steps as four operators for composing features, encoding variability, resolving variability, and generic processing of artifacts. As stated by the authors themselves, the PLA model is helpful when describing complex analyses and designing new ones. Indeed, the PLA model was a source of inspiration for designing our analysis techniques as reusable analysis steps. However, we found the proposed operators to be too generic to be useful in our formal setup. In this sense, our work complements the work by von Rhein et al. [2013] with a formally defined relation among analyses and

intermediate steps, albeit focused on the reliability analysis domain.

Conceptual models and taxonomy: Thüm et al. [2014] established the taxonomy for product-line analyses upon which we based our work, that is, the classification of analysis techniques in three basic strategies (product-based, feature-based, and family-based) and combinations thereof. von Rhein et al. [2013] laid these strategies as dimensions in a cube, meaning analysis strategies can be expressed as a combination of the number of analyzed products (*sampling* dimension), the granularity of feature combinations (*feature grouping* dimension), and the extent to which variability is preserved or resolved during analysis (*variability encoding* dimension). Given that sampling is a matter of restricting possible configurations, and that we prove that our techniques are sound configuration-wise, our work also covers the sampling dimension. Coverage of the two other dimensions must be investigated after formalizing our feature-based strategies.

Meinicke et al. [2014] recently surveyed existing product-line analysis tools and categorized them along four criteria: product-line implementation technique (annotation-based *versus* composition-based approach), analysis technique (e.g., testing, type checking, model checking), strategies for product-line analysis (i.e., the analysis strategies taxonomy by Thüm et al. [2014]), and strategy of the tool (product-based, variability-aware, and variability-encoding). Using this taxonomy, our implemented techniques cover all possibilities on the dimensions of strategies for product-line analysis and strategy of the tool. The dimension of analysis technique would be fixed to model checking for reliability analysis, and the dimension of implementation technique would be constrained to annotation-based UML models.

5.2 Future Work

We are currently in the process of formalizing compositional behavioral models of product lines, as well as the strategies related to the feature-based dimension of the taxonomy and their corresponding soundness proofs. Hence, future work shall present these additional results, thereby formally establishing the commutativity of the diagram in Figure 4.7 as a whole.

This work lays a formal foundation to relate reliability analysis strategies, but we make no claims about the extent to which our results can be generalized to other types of analysis. Thus, future work may extend our analysis theory with product-line analyses other than reliability, seeking commonalities in definitions and soundness proofs. As suggested by Figure 4.7, we believe that category theory can be leveraged to analyze and describe such extended theories, as a means towards the broader goal of finding a set of general principles relating different dimensions of product-line analysis.

During our research, we identified and addressed potential threats to validity. For instance, to mitigate the risk of human error in our proofs, we have submitted them to review by external collaborators. This risk can be further reduced by mechanizing our specifications and proofs using a proof assistant such as PVS [Owre et al., 2001].

Moreover, we have addressed the risk that our theory does not correspond to our implementation by using functional programming principles to develop ReAna-SPL. However, our tool was implemented using Java, which, being an imperative language, provides limited support for the functional paradigm. Thus, we plan to rewrite ReAna-SPL using a functional programming language, such as Haskell, to achieve better correspondence to the theoretical assets. During this refactoring process, we expect to also implement the analysis strategies that we have discovered after the formal specification phase.

Using our product line of reliability analysis tools, future empirical work can compare analysis strategies in search for selection criteria. A complexity analysis of the implementation can also be performed to complement the empirical data with analytical information to justify the results. This knowledge could be helpful as a guide for practitioners, providing an objective view of the trade-offs of using each strategy to analyze a product line according to its attributes.

Bibliography

Rodrigo B Almeida and Paulo Borba. Modeling scenario variability as crosscutting mechanisms. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD)*, pages 125–136, 2009. ISBN 1605584428. doi: 10.1145/1509239.1509258. 12

Vander Alves, Jr. Matos, Pedro, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. In *Transactions on Aspect-Oriented Software Development IV*, volume 4640 of *Lecture Notes in Computer Science*, pages 117–142. Springer, 2007. ISBN 978-3-540-77041-1. doi: 10.1007/978-3-540-77042-8_5. 10

Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lawrence, KS, USA, November 6-10, 2011*, pages 372–375. IEEE Computer Society, 2011. ISBN 978-1-4577-1638-6. doi: 10.1109/ASE.2011.6100075. 33, 42, 52, 73

Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer, 2013a. ISBN 978-3-642-37520-0. doi: 10.1007/978-3-642-37521-7. 1, 6, 7, 9, 10, 12, 38, 48

Sven Apel, Alexander Von Rhein, Philipp Wendler, Armin Groslinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 482–491, Piscataway, NJ, USA, 2013b. IEEE Press. ISBN 9781467330763. doi: 10.1109/ICSE.2013.6606594. 14, 33, 71, 73

John Backus and John. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8): 613–641, aug 1978. ISSN 00010782. doi: 10.1145/359576.359579. 25

R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997. doi: 10.1023/A:1008699807402. 20, 22

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499. 1, 13, 15

Victor R. Basili. The experimental paradigm in software engineering. In *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 3–12, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-57092-6. 24, 26

D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, Jun. 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.23. 11

Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. SPL^{LIFT}: statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 355–364, 2013. doi: 10.1145/2491956.2491976. 1, 74

Sheng Chen and Martin Erwig. Type-based parametric analysis of program families. *ACM SIGPLAN Notices*, 49(9):39–51, aug 2014. ISSN 03621340. doi: 10.1145/2692915.2628155. 74

Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. Family-based modeling and analysis for probabilistic systems - featuring ProFeat. In *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 9633 of *Lecture Notes in Computer Science*, pages 287–304. Springer, 2016. doi: 10.1007/978-3-662-49665-7_17. 2, 16, 23, 46, 71, 73

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer. ISBN 3-540-11212-X. doi: 10.1007/BFb0025774. 72

A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering*, 39 (8):1069–1089, 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.86. 2, 34, 71

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, volume 1, page 335, New York, New York, USA, 2010. ACM Press. ISBN 9781605587196. doi: 10.1145/1806799.1806850. 1, 47

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 321–330. ACM, 2011. doi: 10.1145/1985793.1985838. 1, 2, 71

Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming*, 80, Part B:416–439, Fev. 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2013.09.019. 2, 71

Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001. 1, 7

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-30977-7. 1, 7, 9

Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM, 2006. doi: 10.1145/1173706.1173738. 9

Conrado Daws. Symbolic and parametric model checking of discrete-time Markov chains. In Zhiming Liu and Keijiro Araki, editores, *Proceedings of the First International Conference on Theoretical Aspects of Computing (ICTAC)*, volume 3407 of *Lecture Notes in Computer Science*, pages 280–294, Berlin, Heidelberg, sep 2005. Springer. ISBN 978-3-540-25304-4. doi: 10.1007/b107116. 16

E W Dijkstra. On program families. In *Notes on Structured Programming*. Academic Press, 1971. 7

Dominik Domis, Rasmus Adler, and Martin Becker. Integrating variability and safety analysis models using commercial UML-based tools. In *Proceedings of the 19th International Software Product Line Conference (SPLC)*, pages 225–234. ACM, 2015. ISBN 978-1-4503-3613-0. doi: 10.1145/2791060.2791088. 1

Frank Dordowsky, Richard Bridges, and Holger Tschöpe. Implementing a software product line for a complex avionics system. In *Proceedings of the 15th International Conference on Software Product Lines (SPLC)*, pages 241–250. IEEE, 2011. ISBN 978-1-4577-1029-2. doi: 10.1109/SPLC.2011.11. 1

Clemens Dubslaff, Christel Baier, and Sascha Kluppelholz. Probabilistic model checking for feature-oriented systems. In *Transactions on Aspect-Oriented Software Development XII*, number 8989 in Lecture Notes in Computer Science, pages 180–220. Springer, 2015. ISBN 978-3-662-46733-6 978-3-662-46734-3. doi: 10.1007/978-3-662-46734-3_5. 2, 23, 71, 73

A. Ferreira Leite, V. Alves, G. Nunes Rodrigues, C. Tadonki, C. Eisenbeis, and A.C. Magalhaes Alves de Melo. Automating resource selection and configuration in interclouds through a software product line method. In *Proceedings of the IEEE 8th International Conference on Cloud Computing (CLOUD)*, pages 726–733, Jun. 2015. doi: 10.1109/CLOUD.2015.101. 40

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Boston, MA, USA, 1995. ISBN 0-201-63361-2. 37

Carlo Ghezzi and Amir Molzam Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Information and Software Technology*,

55(3):508–524, Mar. 2013. ISSN 09505849. doi: 10.1016/j.infsof.2012.07.017. 2, 16, 27, 32, 33, 46, 47, 71

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991. ISSN 0360-0300. doi: 10.1145/103162.103163. 40

Lars Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 31–40, New York, NY, USA, 2008. ACM. doi: 10.1145/1368088.1368094. 2, 14

Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta modeling for software architectures. In *MBEES*, pages 1–10, 2011. 12

Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Bernhard Rumpe, Klaus Müller, and Ina Schaefer. Engineering delta modeling languages. In *Proceedings of the 17th International Software Product Line Conference on (SPLC)*, page 22, New York, New York, USA, 2013. ACM Press. ISBN 9781450319683. doi: 10.1145/2491627.2491632. 11, 52

Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. Param: A model checker for parametric Markov models. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 660–664, 2010. doi: 10.1007/978-3-642-14295-6_56. 15, 29, 31, 70

Ernst Moritz Hahn, Holger Hermanns, and Lijun Zhang. Probabilistic reachability for parametric Markov models. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(1):3–19, 2011. doi: 10.1007/s10009-010-0146-x. ix, 16, 17, 18, 19, 58, 59, 70

Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994. ISSN 1433-299X. doi: 10.1007/BF01211866. 14, 72

Ruben Heradio, Hector Perez-Morago, David Fernandez-Amoros, Francisco Cabrerizo Javier, and Enrique Herrera-Viedma. A bibliometric analysis of 20 years of research on software product lines. *Information and Software Technology*, 72:1–15, Abr. 2016. doi: 10.1016/j.infsof.2015.11.004. 1

Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002. ISBN 0898715210. 41

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Relatório técnico, Carnegie-Mellon University Software Engineering Institute, November 1990. 8

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 13th international Conference on Software Engineering (ICSE)*, page 311, New York, New York, USA, 2008. ACM Press. ISBN 9781605580791. doi: 10.1145/1368088.1368131. 48

Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. *ACM SIGPLAN Notices*, 46(10):805, oct 2011. ISSN 03621340. doi: 10.1145/2076021.2048128. 33

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. doi: 10.1007/BFb0053381. 11

Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. A comparison of product-based, feature-based, and family-based type checking. *ACM SIGPLAN Notices*, 49(3):115–124, mar 2014. ISSN 03621340. doi: 10.1145/2637365.2517213. 74

Matthias Kowal, Ina Schaefer, and Mirco Tribastone. Family-based performance analysis of variant-rich software systems. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, pages 94–108, New York, NY, USA, 2014. Springer. ISBN 978-3-642-54803-1. doi: 10.1007/978-3-642-54804-8_7. 73

Matthias Kowal, Max Tschaikowski, Mirco Tribastone, and Ina Schaefer. Scaling size and parameter spaces in variability-aware software performance models. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 407–417, Nov. 2015. doi: 10.1109/ASE.2015.16. 2, 71, 73

J. Kramer, J. Magee, M. Sloman, and A. Lister. CONIC: an integrated approach to distributed computer control systems. *IEE Proceedings E - Computers and Digital Techniques*, 130(1), Jan. 1983. doi: 10.1049/ip-e.1983.0001. 40

Charles W. Krueger. Easing the transition to software mass customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering (PFE)*, pages 282–293, London, UK, UK, 2002. Springer. ISBN 3-540-43659-6. 10, 26

M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editores, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. doi: 10.1007/978-3-642-22110-1_47. 15, 70

Jeremy T. Lanman, Rowland Darbin, Jorge Rivera, Paul C. Clements, and Charles W. Krueger. The challenges of applying service orientation to the U.S. army's live training software product line. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 244–253. ACM, 2013. ISBN 978-1-4503-1968-3. doi: 10.1145/2491627.2491649. 1

Gary T. Leavens and Yoonsik Cheon. Design by contract with jml. Available at `http://www.jmlspecs.org`, 2006. 13

Harry C. Li, Shriram Krishnamurthi, and Kathi Fisler. Modular verification of open features using three-valued model checking. *Automated Software Engineering*, 12(3): 349–382, Jul. 2005. ISSN 0928-8910. doi: 10.1007/s10515-005-2643-9. 72

Jing Liu, Samik Basu, and Robyn R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18 (1):39–76, Dez. 2010. ISSN 0928-8910. doi: 10.1007/s10515-010-0075-7. 72

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. An overview on analysis tools for software product lines. In *Proceedings of the 18th International Software Product Line Conference (SPLC)*, pages 94–101, New York, New York, USA, sep 2014. ACM Press. ISBN 9781450327398. doi: 10.1145/2647908.2655972. 75

Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Out. 1992. ISSN 0018-9162. doi: 10.1109/2.161279. 14

Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. Systematic derivation of correct variability-aware program analyses. *Science of Computer Programming*, 105:145–170, Jul. 2015. ISSN 01676423. doi: 10.1016/j.scico.2015.04.005. 2, 23, 74

V. Nunes, P. Fernandes, V. Alves, and G. Rodrigues. Variability management of reliability models in software product lines: An expressiveness and scalability analysis. In *Proceedings of the Sixth Brazilian Symposium on Software Components Architectures and Reuse (SBCARS)*, pages 51–60, Set. 2012. doi: 10.1109/SBCARS.2012.23. 2, 71

V. Nunes, D. Mendonça, G. Rodrigues, and V. Alves. Towards compositional approach for parametric model checking in software product lines. In *Proceedings of the International Workshop on Architecting Dependable Systems (WDAS)*. SBC, 2013. 46

Object Management Group. The UML profile for MARTE: Modeling and analysis of real-time and embedded systems. Available at http://www.omg.org/spec/MARTE/1.1/, 2011. Version 1.1. 27

S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, Nov. 2001. 12, 13, 76

Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. Coevolution of variability models and related artifacts: A case study from the linux kernel. In *Proceedings of the 17th International Software Product Line Conference (SPLC)*, pages 91–100, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1968-3. doi: 10.1145/2491627.2491628. 11

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1. 13

Malte Plath and Mark Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, Set. 2001. doi: 10.1016/S0167-6423(00)00018-6. 40

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Secaucus, NJ, USA, 2005. ISBN 3540243720. 1, 7

Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 347–350. IEEE Computer Society, 2008. doi: 10.1109/ASE.2008.45. 34, 42, 73

Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997. doi: 10.1007/BFb0053389. 11

Genaína Nunes Rodrigues, Vander Alves, Vinicius Nunes, André Lanna, Maxime Cordy, Pierre-Yves Schobbens, Amir Molzam Sharifloo, and Axel Legay. Modeling and verification for probabilistic properties in software product lines. In *Proceedings of the 16th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, pages 173–180. IEEE Computer Society, 2015. doi: 10.1109/HASE.2015.34. ix, 1, 2, 8, 9, 16, 27, 40, 72

W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering (ICSE)*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-89791-216-0. 10

Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines (SPLC)*, pages 77–91, Berlin, Heidelberg, 2010. Springer. ISBN 3-642-15578-2, 978-3-642-15578-9. 12

Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A systematic mapping study of software product lines testing. *Information and Software Technology*, 53(5): 407–423, Mai. 2011. ISSN 09505849. doi: 10.1016/j.infsof.2010.12.003. 12

Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the linux kernel a software product line? In *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL)*, 2007. 7

Dag I. K. Sjøberg, Tore Dybå, Bente C. D. Anda, and Jo E. Hannay. *Building Theories in Software Engineering*, In *Guide to Advanced Empirical Software Engineering*, pages 312–336. Springer, London, 2008. ISBN 978-1-84800-044-5. doi: 10.1007/978-1-84800-044-5_12. 24

Leopoldo Teixeira, Vander Alves, Paulo Borba, and Rohit Gheyi. A product line of theories for reasoning about safe evolution of product lines. In *Proceedings of the 19th International Conference on Software Product Line (SPLC)*, pages 161–170, New York, New York, USA, Jul. 2015. ACM Press. ISBN 9781450336130. doi: 10.1145/2791060.2791105. 12

Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof composition for deductive verification of software product lines. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 270–277. IEEE, Mar. 2011. ISBN 978-1-4577-0019-4. doi: 10.1109/ICSTW.2011.48. 14

Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-based deductive verification of software product lines. *ACM SIGPLAN Notices*, 48(3):11–11–20–20, Abr. 2013. ISSN 0362-1340. doi: 10.1145/2480361.2371404. 52

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):1–45, Jun. 2014. ISSN 03600300. doi: 10.1145/2580950. 2, 3, 4, 12, 13, 23, 24, 26, 29, 32, 33, 55, 58, 59, 70, 71, 72, 75

Lucineia Turnes, Rodrigo Bonifácio, Vander Alves, and Ralf Lammel. Techniques for developing a product line of product line tools: A comparative study. In *2011 Fifth Brazilian Symposium on Software Components, Architectures and Reuse*, pages 11–20. IEEE, Set. 2011. ISBN 978-0-7695-4626-1. doi: 10.1109/SBCARS.2011.13. 11, 12

Otto von Guericke University of Magdeburg. SPL2go. Available at `http://spl2go.cs.ovgu.de/`. Accessed: 2016-01-27. 40

Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, Secaucus, NJ, USA, 2007. ISBN 3540714367. 1, 7

J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 45–54. IEEE Comput. Soc, Ago. 2001. ISBN 0-7695-1360-3. doi: 10.1109/WICSA.2001.948406. 7

Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. The PLA model. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, page 1, New York, New York, USA, Jan. 2013. ACM Press. ISBN 9781450315418. doi: 10.1145/2430502.2430522. 24, 74, 75

Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming*, 85(1):125–145, jan 2016. ISSN 23522208. doi: 10.1016/j.jlamp.2015.06.007. 2, 23, 33, 34, 42, 52, 73, 74

Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational data structures. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, pages 213–226, New York, New York, USA, oct 2014. ACM Press. ISBN 9781450332101. doi: 10.1145/2661136.2661143. 30, 62

David M. Weiss. The product line hall of fame. In *Proceedings of the 12th International Software Product Line Conference (SPLC)*, page 395, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3303-2. doi: 10.1109/SPLC.2008.56. 1

Appendix A

Probabilistic Models

This appendix presents the probabilistic models of the beverage machine product line example (Section 4.1) in their entirety.

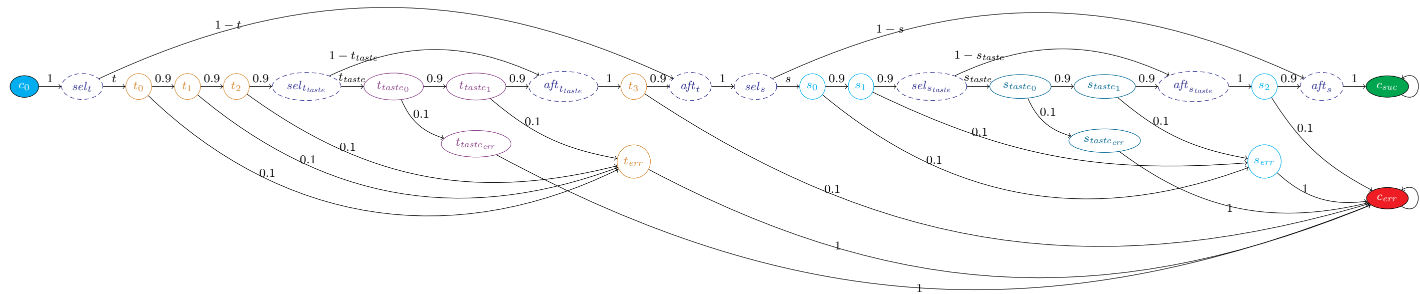
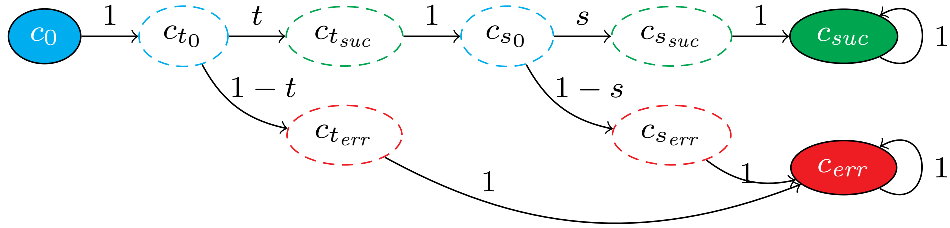
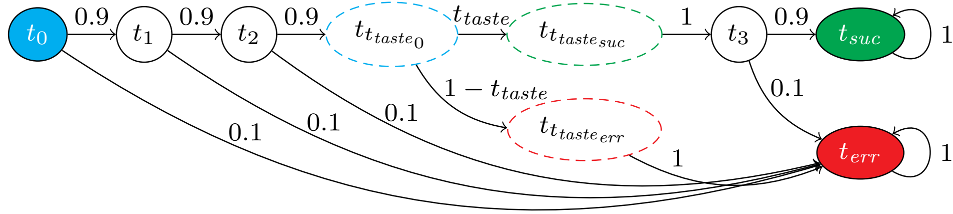


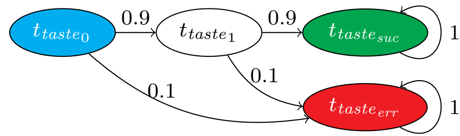
Figure A.1: Complete annotative PMC for the vending machine.



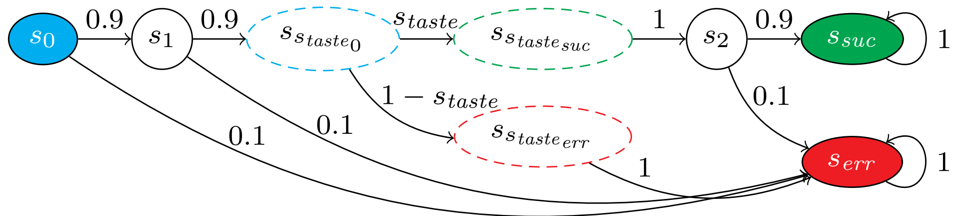
(a) Top-level compositional PMC for the vending machine (common behavior and main variation points).



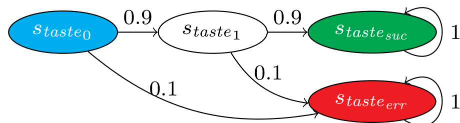
(b) Compositional PMC for the behavior of serving tea.



(c) Compositional PMC for the behavior of adding taste to tea.



(d) Compositional PMC for the behavior of serving soda.



(e) Compositional PMC for the behavior of adding taste to soda.

Figure A.2: Compositional PMCs for the vending machine.

Appendix B

Theory Dependencies

This appendix is a compilation of dependency graphs for the main theorems presented in this work. These directed graphs are depicted in diagrams where nodes represent theory elements, while edges denote the source element depends on the target element. Dependencies indicate that the statement of the element at hand makes use of other definitions, or that its proof (if it is a theorem or lemma) relies on the element on which it depends. Element names are colored according to their types: theorems are `cyan`, lemmas are `green`, and definitions and properties are `red`.

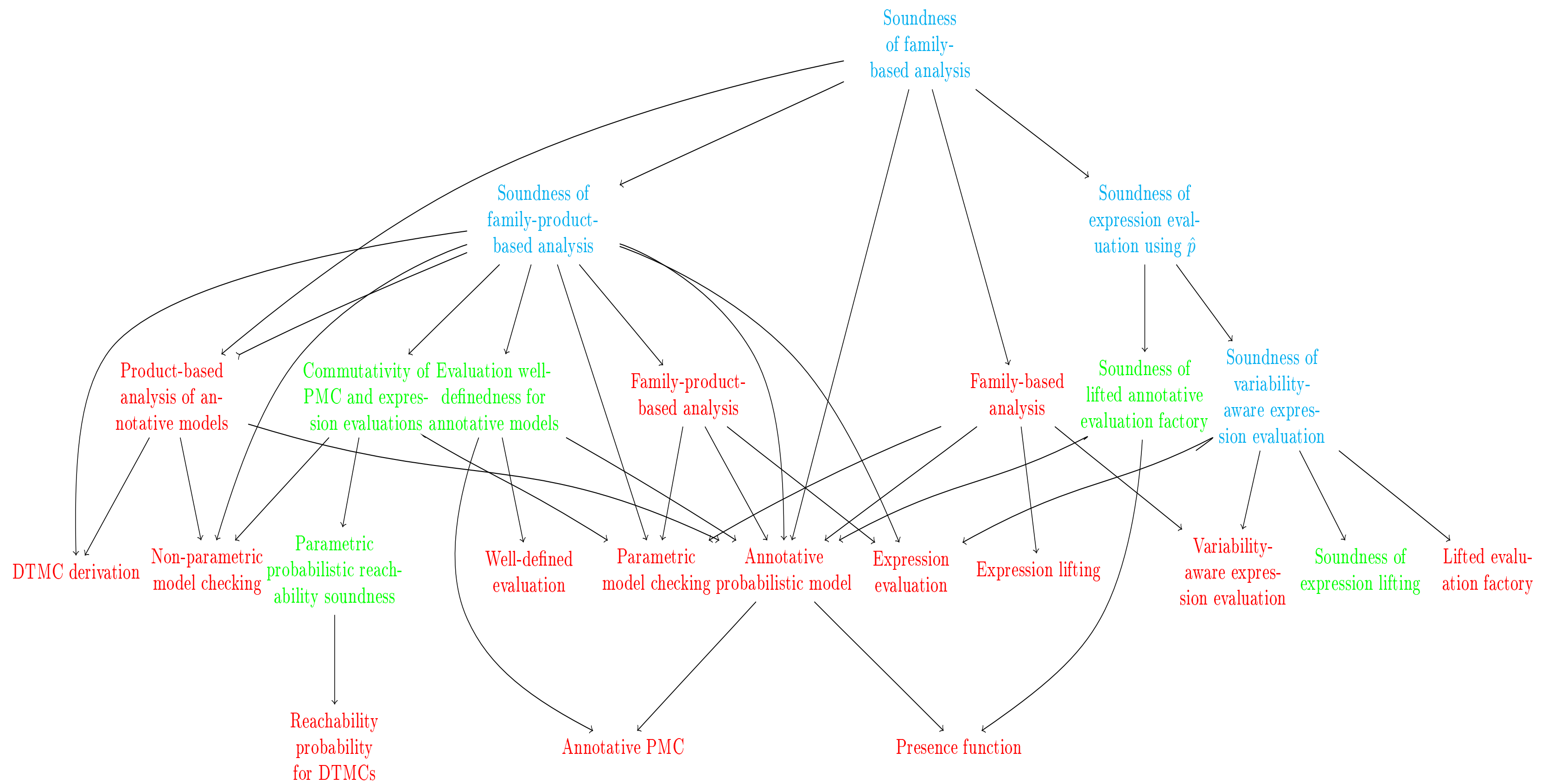


Figure B.1: Overall theory structure.

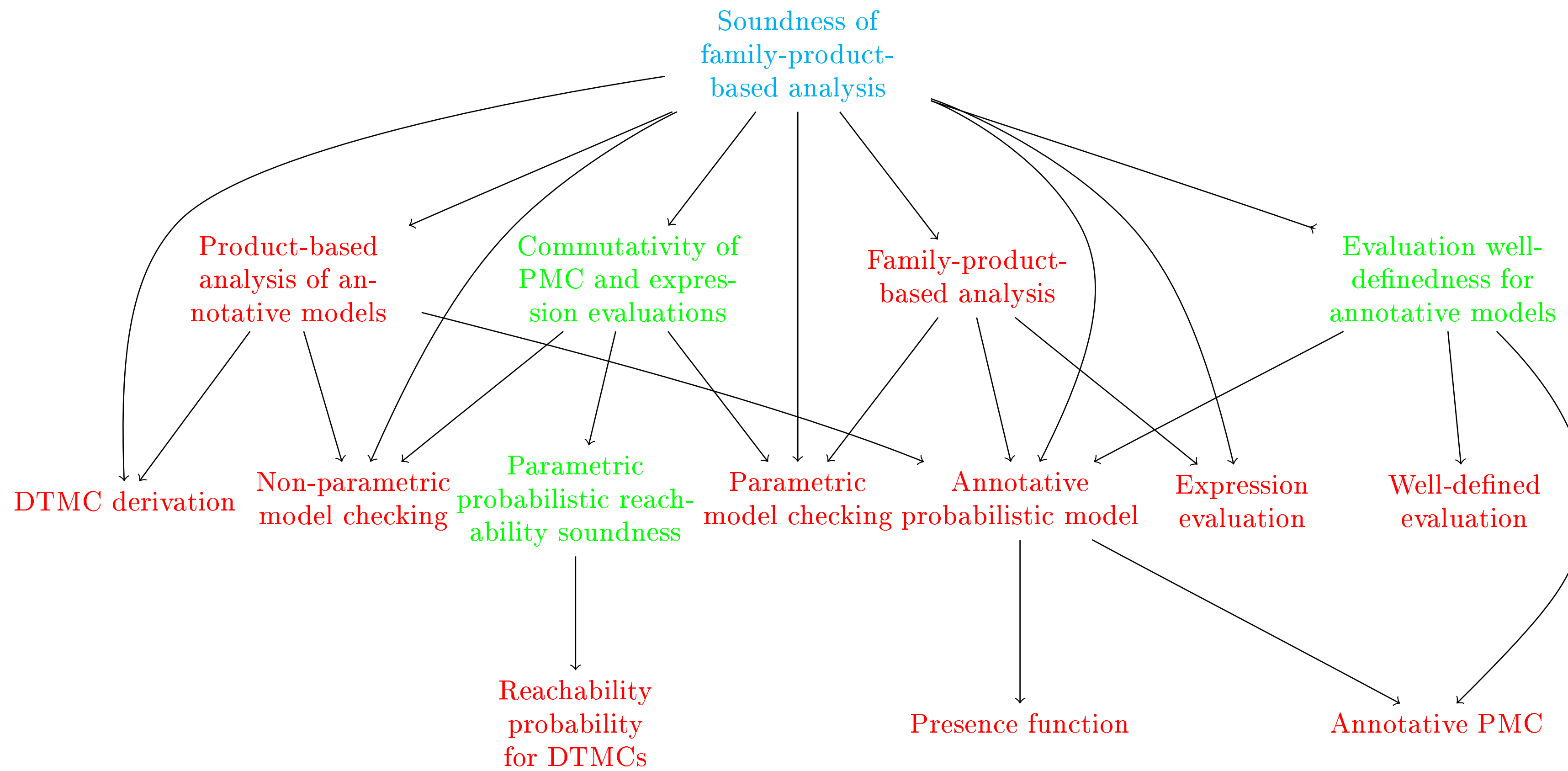


Figure B.2: Dependencies for Theorem 1 (Soundness of family-product-based analysis).

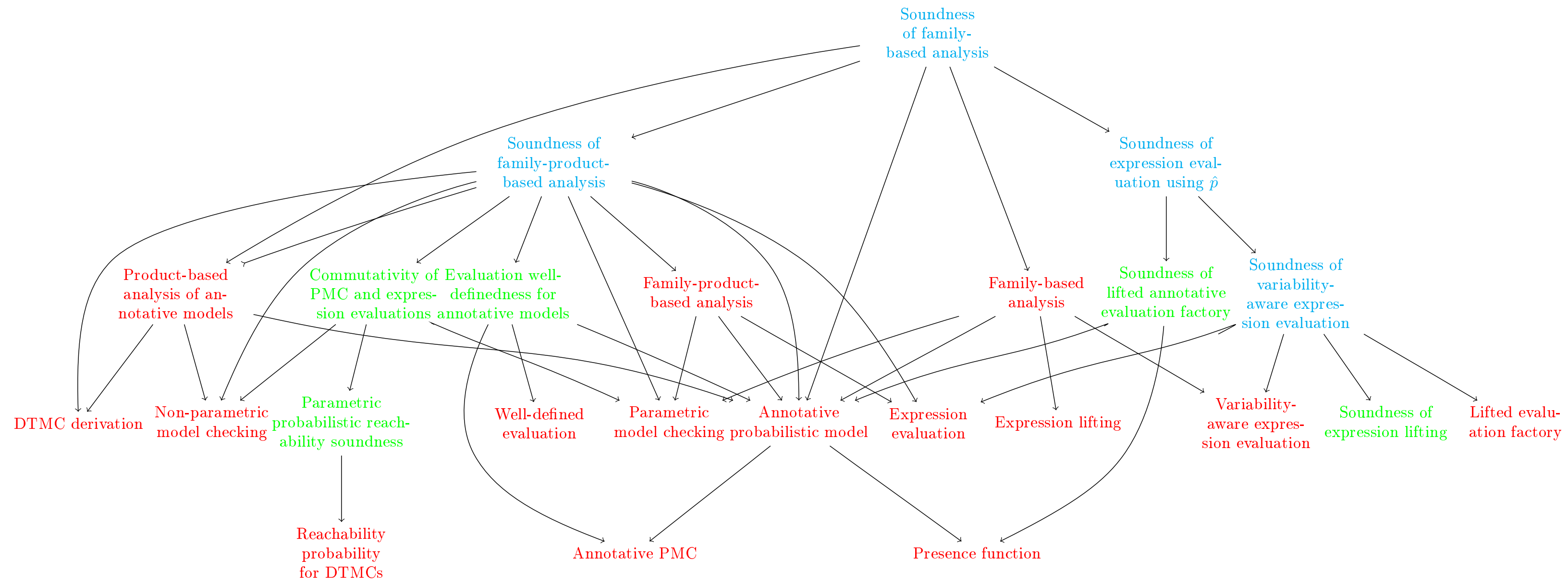


Figure B.3: Dependencies for Theorem 4 (Soundness of family-based analysis).