Universidade de Brasília

Instituto de Ciências Exatas

Departamento de Ciência da Computação


# A Machine-Verified Theory of Commuting Strategies for Product-Line Reliability Analysis


Thiago Mael de Castro



Brasília

2019

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


# A Machine-Verified Theory of Commuting Strategies for Product-Line Reliability Analysis


Thiago Mael de Castro


Tese apresentada como requisito parcial
para conclusão do Doutorado em Informática

Orientador
Prof. Dr. Vander Ramos Alves

Brasília
2019

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

CC355m

Castro, Thiago Mael de
A Machine-Verified Theory of Commuting Strategies for Product-Line Reliability Analysis / Thiago Mael de Castro; orientador Vander Ramos Alves. -- Brasília, 2019.
230 p.

Tese (Doutorado - Doutorado em Informática) -- Universidade de Brasília, 2019.

1. Software product lines. 2. Reliability analysis. 3. Model checking. 4. Formal verification. 5. Interactive theorem proving. I. Alves, Vander Ramos, orient. II. Título.

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

# A Machine-Verified Theory of Commuting Strategies for Product-Line Reliability Analysis

Thiago Mael de Castro

Tese apresentada como requisito parcial
para conclusão do Doutorado em Informática

Prof. Dr. Vander Ramos Alves (Orientador)
Universidade de Brasília

Prof.ª Dr.ª Genaína Nunes Rodrigues
Universidade de Brasília

Dr. Maurice H. ter Beek
Consiglio Nazionale delle Ricerche

Prof. Dr. Rohit Gheyi
Universidade Federal de Campina Grande

Prof.ª Dr.ª Genaína Nunes Rodrigues
Coordenadora do Doutorado em Informática

Brasília, 13 de dezembro de 2019

To all the people who (indirectly) supported this research.

# Acknowledgements

To my beloved wife, Isabela, who is able to spot frying neurons from a distance. Thank you for supporting me whenever I stumbled and for helping me drop the keyboard when I needed some fresh air. This work is yours as much as it is mine.

Thanks to all my families—by birth and by choice (not mutually exclusive)—for all the inspiration and support throughout this amazing journey of life and work.

Thanks to my supervisors, Vander and Leopoldo, for trusting me and for understanding my limitations. This research would not be possible without your precise and reliable guidance, and this researcher would not be possible without your respect and the example you set. I am really proud of having you as mentors, and I will make sure to carry on the lessons I learned from you.

Thanks to all the present and past generations of students within our product-line research group, for being the shoulders on which I stand. In particular, I will be forever grateful to André Lanna and our discussions in the lab, which eventually led to a certain theory of commuting strategies for product-line analysis.

Thanks to Sven Apel and Pierre-Yves Schobbens for the rigorous and precise contributions that established the foundations of this work. I would also like to thank the members of the qualifying and defense examination boards—Genaína Rodrigues, Maurice ter Beek, Rohit Gheyi, and Alexandre Mota—for the detailed feedback and for the extremely interesting discussions about related and future work.

Last, but not least, I would like to thank my coworkers from the past 12 years. You all contributed in some way to myself as a person and as an engineer; by induction, you also contributed to me as a researcher. Special thanks to the C2 development team; your autonomy, expertise, and selfless dedication assured me that our mission would be accomplished no matter what, so that I could focus my out-of-office time in this research.

*"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 Expandido

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. Mesmo linhas de produtos com dezenas ou centenas de opções de configuração (*features*) podem dar origem a milhões de produtos, tornando inviável verificar a qualidade de cada um desses produtos isoladamente. Não obstante, linhas de produtos de software crítico (por exemplo, nos domínios de aviação e sistemas médicos) necessitam garantir que seus produtos são confiáveis.

Existem diversas abordagens *cientes de variabilidade* para análise de linha de produtos, as quais adaptam técnicas de análise de produtos isolados para lidar com variabilidade de forma eficiente. Tais abordagens podem ser classificadas em três dimensões de análise: *product-based* (os objetos de análise são produtos ou modelos destes), *family-based* (apenas artefatos de domínio e combinações válidas são verificados) e *feature-based* (artefatos de domínio que implementam uma dada *feature* são analisados isoladamente) [85]. Mais de uma dimensão pode ser combinada em uma mesma técnica, dando origem a análises *feature-family-based* (*features* são parcialmente analisadas isoladamente, depois combinam-se os resultados intermediários de maneira *family-based*) ou *family-product-based* (artefatos de domínio são parcialmente analisados considerando-se apenas configurações válidas, culminando em resultados que podem, então, ser submetidos a análise enumerativa), por exemplo.

Essas estratégias combinadas possuem vantagens e desvantagens distintas, as quais variam de acordo com a técnica de análise em questão. Por esse motivo, há estudos empíricos que avaliam as relações de compromisso específicas a cada técnica [49, 54, 59, 92]. Além disso, a utilização correta e automática (ou mesmo sistemática) de técnicas para análise de software consagradas em linhas de produtos ainda é uma questão de pesquisa não respondida. Assim, a corretude de técnicas para análise de linhas de produtos precisa ser demonstrada caso a caso.

Abordagens existentes para análise de linhas de produtos frequentemente estendem técnicas de análise de software tradicionais (ou seja, aplicáveis a produtos independentes) para operar em linhas de produtos de maneira *family-based* [14, 19, 34, 62, 91]. Então, demonstra-se a corretude de tais abordagens por comparação à estratégia *product-based* correspondente. No contexto de análise de confiabilidade, particularmente, 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 umas às outras. A falta de uma teoria com essas propriedades dificulta que se raciocine formalmente sobre o relacionamento entre as dimensões de análise e técnicas de análise derivadas.

De fato, é fundamental provar que um método de análise produz resultados corretos, especialmente para sistemas críticos. Por exemplo, Lanna et al. [54] propuseram uma estratégia *feature-family-based* para análise de confiabilidade de linhas de produtos sob o ponto de vista do usuário. Essa abordagem foi avaliada empiricamente, e os resultados indicam que possui melhor desempenho que as técnicas previamente existentes, tanto em relação ao tempo quanto ao uso de memória. Particularmente no caso de uma linha de produtos crítica de redes de sensores do corpo humano [70] (com 16 *features* e 298 configurações possíveis), observou-se que a maior parte das estratégias existentes seria inviável se mais que 5 novas *features* fossem adicionadas em versões futuras [54]. Entretanto, a falta de evidência de que as diferentes estratégias são mutuamente equivalentes limita os resultados desses estudos empíricos existentes.

Para ajudar a preencher essa lacuna, este trabalho investiga a corretude das estratégias de análise avaliadas por Lanna et al. [54]—ou seja, técnicas de análise de confiabilidade *orientada ao usuário*, que operam a partir de técnicas de *model checking* aplicadas a modelos baseados em cadeias de Markov de tempo discreto (*Discrete-time Markov Chains*—DTMC). Correspondentemente, utilizamos uma definição de confiabilidade como a probabilidade de execuções dos modelos em questão alcançarem estados que denotam sucesso. Embora tais modelos não sejam genéricos a ponto de representar *qualquer* sistema, eles assumem premissas que contemplam linhas de produtos de interesse, como redes de sensores do corpo humano.

Nesse contexto, formalizamos sete abordagens para análise de confiabilidade em linhas de produtos, cobrindo todas as três dimensões de análise e incluindo a primeira instância de análise *feature-family-product-based* na literatura [15]. Provamos que as estratégias formalizadas são corretas em relação à abordagem para análise de confiabilidade de produtos individuais, fortalecendo as comparações empíricas entre elas. Desse modo, engenheiros podem escolher a estratégia mais apropriada à linha de produtos em questão, seguros de sua corretude.

A formalização aqui apresentada parte de uma caracterização matemática de modelos de linhas de produtos *composicionais* como um conjunto finito de modelos de linhas de produtos *anotativas*, o qual é dotado de uma estrutura conferida por uma relação bem-fundada de dependência entre os modelos. Tendo modelos composicionais ou anotativos, pode-se optar por derivar modelos sem variabilidade (representando produtos individuais), cuja confiabilidade é, então, calculada por meio de *model checking*. Alternativamente, pode-se aplicar *model checking* paramétrico, o que resulta em expressões algébricas que representam a confiabilidade dos modelos em função das opções de configuração. As expressões resultantes podem ser valoradas de forma enumerativa (ou seja, para cada configuração possível); como alternativa, podemos ressignificar a semântica das expressões (*lifting*) para trabalhar com diagramas de decisão algébricos (*Algebraic Decision Diagrams*—ADD). Utilizando ADD, é possível realizar eficientemente operações aritméticas cuja semântica equivale a enumerar todas as configurações possíveis.

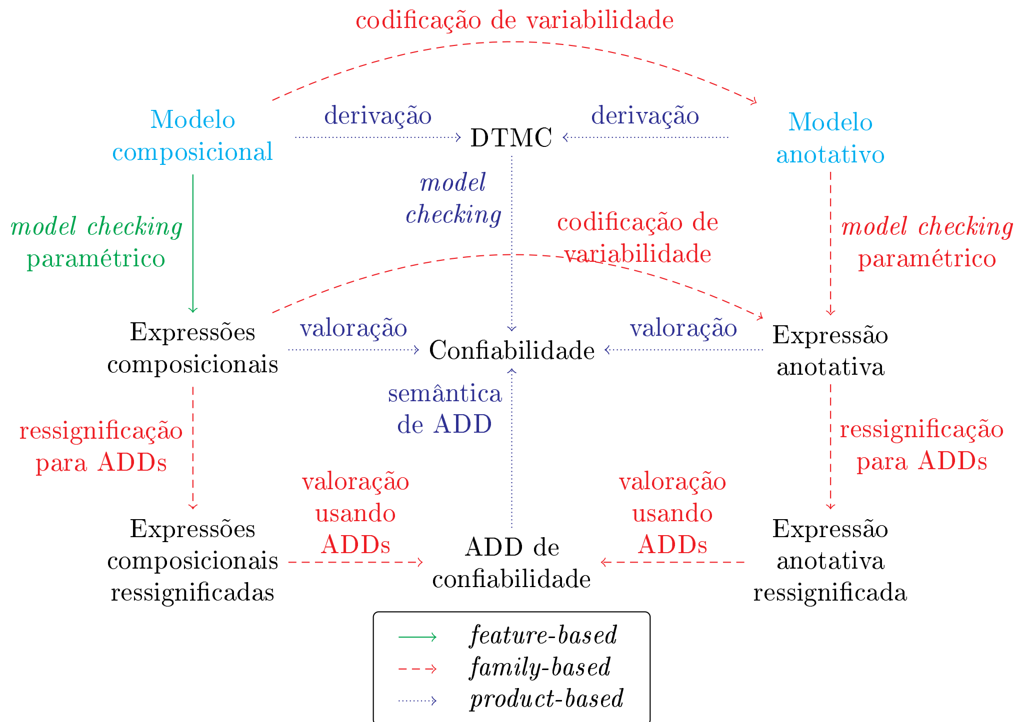



Figura 1: Diagrama comutativo de estratégias para análise de confiabilidade em linhas de produtos (versão resumida)

Adicionalmente, apresentamos essas opções alternativas em um diagrama comutativo de passos intermediários de análise (Figura 1), o qual relaciona estratégias diferentes e permite reusar demonstrações de corretude entre elas. Tal diagrama representa graficamente as possíveis composições de funções, ilustrando objetivamente a dimensão de análise de linha de produtos utilizada (por meio da cor e da forma das setas). Essa visão contribui para uma compreensão mais abrangente sobre os princípios subjacentes às estratégias, o

que visualiza-se poder ajudar outros pesquisadores a alçar técnicas de análise de software para abordagens cientes de variabilidade ainda inexploradas.

Além disso, reduzimos o risco de erro humano por meio da mecanização da teoria resultante no provador interativo de teoremas chamado PVS (*Prototype Verification System*). Essa ferramenta permite a especificação formal de definições, lemas e teoremas, além de possibilitar automação e verificação das demonstrações. Como resultado do esforço de mecanização, identificamos erros e imprecisões na versão manualmente especificada de nossa teoria, os quais foram consequentemente corrigidos.

Com isso, além de aumentar a confiança na corretude de nossos resultados, disponibilizamos uma teoria verificada por máquina potencialmente reutilizável. Em particular, parte das especificações e teoremas desenvolvidos em PVS referem-se a tópicos de interesse mais amplo, como DTMC, expressões algébricas, ADD, conjuntos e listas; portanto, estão em processo de submissão para análise e integração à biblioteca PVS da NASA. Ademais, documentamos as lições aprendidas durante o processo de mecanização, as quais serão submetidas à apreciação da comunidade científica.

Em adição às contribuições aqui apresentadas, espera-se que este trabalho, em longo prazo, apoie a construção do arcabouço para uma teoria geral de análise em linhas de produtos. Dessa forma, almeja-se contribuir para o problema mais amplo de alçar técnicas de análise de software para operar em linhas de produtos de maneira correta por construção.

**Palavras-chave:** Linhas de produtos de software, Análise de confiabilidade, Model checking, Verificação formal, Prova interativa de teoremas

# 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. Nonetheless, product lines of safety-critical software (e.g., in the domains of avionics and medical systems) need to ensure that its products are reliable.

There are 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 composable 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 hinders formal reasoning on the relationship between the analysis dimensions and derived analysis techniques. Moreover, as long as there is no evidence that the different examined strategies are mutually equivalent, the existing empirical studies comparing them will have limited results.

To address this issue, we formalize seven approaches to user-oriented reliability analysis of product lines, covering all three analysis dimensions and including the first instance of a feature-family-product-based analysis in the literature. We prove the formalized analysis strategies to be sound with respect to reliability analysis of a single product, thereby strengthening the existing empirical comparison between them.

Furthermore, we present a commuting diagram of intermediate analysis steps, which relates different strategies and enables the reuse of soundness proofs between them. Such view contributes to a more comprehensive understanding of underlying principles used in these strategies, which we envision could help other researchers to lift existing single-product analysis techniques to yet under-explored variability-aware approaches.

Additionally, we reduce the risk of human error by mechanizing the resulting theory in the PVS interactive theorem prover. As a result, we identified and corrected errors and

imprecisions of the handcrafted version. Hence, we document lessons learned throughout the mechanization process and provide a potentially reusable machine-verified theory.

**Keywords:** Software product lines, Reliability analysis, Model checking, Formal verification, Interactive theorem proving

# List of Figures


1 Diagrama comutativo de estratégias para análise de confiabilidade em linhas de produtos (versão resumida) . . . . . . . . . . . . . . . . . . . . . . ix

1.1 Overview of the commutative diagram of product-line reliability analysis strategies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.1 Feature model of the BSN product line [76] . . . . . . . . . . . . . . . . . . . 12
2.2 Example graph view of a DTMC and the corresponding reachability probability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.3 Example graph view of a PMC and the intuition for the corresponding reachability probability expression . . . . . . . . . . . . . . . . . . . . . . . 22
2.4 Elimination of state $s$ in the parametric reachability probability algorithm (adapted from Hahn et al. [41]) . . . . . . . . . . . . . . . . . . . . . . . 23
2.5 Statement of Lemma 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.6 ADD $A_f$ representing the Boolean function $f$ in Equation (2.2) . . . . . . 25
2.7 Example of an arithmetic operation over ADDs . . . . . . . . . . . . . . . . 26
2.8 Example of an `ITE` operation over ADDs . . . . . . . . . . . . . . . . . . . 27
2.9 Alternative ordering for encoding the Boolean function $f$ in Equation (2.2) as an ADD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3.1 Vending machine product line example . . . . . . . . . . . . . . . . . . . 36
3.2 Annotative PMC for the vending machine . . . . . . . . . . . . . . . . . . 38
3.3 Compositional PMCs for the vending machine . . . . . . . . . . . . . . . . 43
3.4 Example of a partial composition of two PMCs . . . . . . . . . . . . . . . 46
3.5 Dependency relation induced in the vending machine . . . . . . . . . . . . 48
3.6 Feature disabler compositional PMC $\mathcal{P}_\perp$ . . . . . . . . . . . . . . . . . . 49
3.7 Commutative diagram of product-line reliability analysis strategies . . . . . 52
3.8 Example of family-product-based analysis ($\hat{\alpha}$ followed by $\sigma$) in contrast to a product-based analysis ($\pi$ followed by $\alpha$) of an annotative PMC, for a configuration satisfying $x$'s presence condition . . . . . . . . . . . . . . . . 56

3.9 Statement of Theorem 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.10 Example of lifted expression evaluation using $\hat{p}$ . . . . . . . . . . . . . . . . . 61
3.11 Statement of Theorem 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
3.12 Alternative views of the statement of Theorem 4 . . . . . . . . . . . . . . . . 63
3.13 Statement of Theorem 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
3.14 Example of lifted compositional expression evaluation . . . . . . . . . . . . 67
3.15 Statement of Theorem 6 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
3.16 Alternative views of the statement of Theorem 7 . . . . . . . . . . . . . . . . 70
3.17 Example `ITE` operator for PMCs . . . . . . . . . . . . . . . . . . . . . . . . 71
3.18 Statement of Theorem 8 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
3.19 Statement of Theorem 9 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
3.20 Commuting diagram leading to the feature-family-product-based strategy . 77
3.21 Statement of Theorem 10 . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

4.1 Proof of Lemma 2 in PVS (left) and Coq (right) . . . . . . . . . . . . . . . 83
4.2 Elimination of state $s_{sink}$ “loses” variable $x$ . . . . . . . . . . . . . . . . . . 112
4.3 Intuition for lemmas regarding state elimination . . . . . . . . . . . . . . . . 114
4.4 Proportion of specification and proof effort . . . . . . . . . . . . . . . . . . 115
4.5 Comparison of PVS theories regarding percentage of mechanized lemmas *versus* percentage of the original lemmas . . . . . . . . . . . . . . . . . . . 117
4.6 Usage of prover commands per category . . . . . . . . . . . . . . . . . . . 122
4.7 Intuition for lemmas regarding state elimination . . . . . . . . . . . . . . . . 135

A.1 Generic PMCs in Lemma 13 . . . . . . . . . . . . . . . . . . . . . . . . . . 166

B.1 Complete annotative PMC for the vending machine . . . . . . . . . . . . . 181
B.2 Compositional PMCs for the vending machine . . . . . . . . . . . . . . . . 182

C.1 Dependencies between PVS theories (direct dependencies that can be deduced from transitivity are filtered for readability) . . . . . . . . . . . . . . 188

D.1 Overall theory structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . 201
D.2 Overall theory structure (only definitions) . . . . . . . . . . . . . . . . . . 202
D.3 Overall theory structure (only theorems and lemmas) . . . . . . . . . . . . 203
D.4 Dependencies for Theorem 1 (Soundness of family-product-based analysis) 204
D.5 Dependencies for Theorem 4 (Soundness of family-based analysis) . . . . . 205
D.6 Dependencies for Theorem 5 (Soundness of feature-product-based analysis) 206
D.7 Dependencies for Theorem 7 (Soundness of feature-family-based analysis) . 207

D.8 Dependencies for Theorem 10 (Soundness of feature-family-product-based analysis) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 208

# List of Tables


1.1 Research outline . . . 8

C.1 Description of PVS theories . . . 183
C.1 Description of PVS theories (continued) . . . 184
C.1 Description of PVS theories (continued) . . . 185
C.1 Description of PVS theories (continued) . . . 186
C.1 Description of PVS theories (continued) . . . 187
C.2 Mapping between the manual and the mechanized specifications . . . 189
C.2 Mapping between the manual and the mechanized specifications (continued) 190
C.2 Mapping between the manual and the mechanized specifications (continued) 191
C.2 Mapping between the manual and the mechanized specifications (continued) 192
C.2 Mapping between the manual and the mechanized specifications (continued) 193
C.2 Mapping between the manual and the mechanized specifications (continued) 194
C.2 Mapping between the manual and the mechanized specifications (continued) 195
C.3 Categories of proof commands . . . 196
C.4 Proof commands . . . 196
C.4 Proof commands (continued) . . . 197
C.4 Proof commands (continued) . . . 198
C.5 PVS specification and coverage of the original theory . . . 199

# List of Definitions


1 Property (Reachability probability for DTMCs) . . . . . . . . . . . . . . . . 20
1 Definition (Parametric Markov Chain) . . . . . . . . . . . . . . . . . . . . 20
2 Definition (Expression evaluation) . . . . . . . . . . . . . . . . . . . . . . 21
3 Definition (Well-defined evaluation) . . . . . . . . . . . . . . . . . . . . . 21
4 Definition (State elimination step) . . . . . . . . . . . . . . . . . . . . . . 23

5 Definition (Annotative PMC) . . . . . . . . . . . . . . . . . . . . . . . . . 38
6 Definition (Presence function) . . . . . . . . . . . . . . . . . . . . . . . . 39
7 Definition (Evaluation factory) . . . . . . . . . . . . . . . . . . . . . . . . 40
8 Definition (Annotative probabilistic model) . . . . . . . . . . . . . . . . . . 40
9 Definition (DTMC derivation) . . . . . . . . . . . . . . . . . . . . . . . . . 41
10 Definition (Compositional PMC) . . . . . . . . . . . . . . . . . . . . . . . 44
11 Definition (Compositional PMC slot) . . . . . . . . . . . . . . . . . . . . . 44
12 Definition (Partial PMC composition) . . . . . . . . . . . . . . . . . . . . . 45
13 Definition (Identifying function) . . . . . . . . . . . . . . . . . . . . . . . 47
14 Definition (Dependency relation induced in compositional PMCs) . . . . . 47
15 Definition (Minimal and maximal compositional PMCs) . . . . . . . . . . . 48
16 Definition (Feature disabler compositional PMC) . . . . . . . . . . . . . . 49
17 Definition (Composition factory) . . . . . . . . . . . . . . . . . . . . . . . 50
18 Definition (Compositional probabilistic model) . . . . . . . . . . . . . . . . 50
19 Definition (Derivation by composition) . . . . . . . . . . . . . . . . . . . . 51
20 Definition (Non-parametric model checking) . . . . . . . . . . . . . . . . . 53
21 Definition (Parametric model checking) . . . . . . . . . . . . . . . . . . . . 55
22 Definition (Expression evaluation) . . . . . . . . . . . . . . . . . . . . . . 55
23 Definition (Expression lifting) . . . . . . . . . . . . . . . . . . . . . . . . . 58
24 Definition (Lifted evaluation factory) . . . . . . . . . . . . . . . . . . . . . 59
25 Definition (Variability-aware expression evaluation) . . . . . . . . . . . . . 60
26 Definition (Compositional evaluation factory) . . . . . . . . . . . . . . . . 64
27 Definition (Lifted compositional evaluation factory) . . . . . . . . . . . . . 67
28 Definition (Variability encoding function for PMCs) . . . . . . . . . . . . . 72

29 Definition (Variability encoding of PMCs) . . . . . . . . . . . . . . . . . . 72
30 Definition (ITE operator for expressions) . . . . . . . . . . . . . . . . . . . 74
31 Definition (Variability encoding function for expressions) . . . . . . . . . . 74
32 Definition (Variability encoding of expressions) . . . . . . . . . . . . . . . . 75

33 Definition (Compositional PMC renaming) . . . . . . . . . . . . . . . . . . 164
34 Definition (Total PMC composition) . . . . . . . . . . . . . . . . . . . . . 164
35 Definition (ITE operator for PMCs) . . . . . . . . . . . . . . . . . . . . . 173

# List of Theorems


1 Lemma (Parametric probabilistic reachability soundness) . . . . . . . . . . 24

2 Lemma (Evaluation well-definedness for annotative models) . . . . . . . . . 41
3 Lemma (Commutativity of PMC and expression evaluations) . . . . . . . . 56
1 Theorem (Soundness of family-product-based analysis) . . . . . . . . . . . 57
4 Lemma (Soundness of expression lifting) . . . . . . . . . . . . . . . . . . . 59
2 Theorem (Soundness of variability-aware expression evaluation) . . . . . . 60
5 Lemma (Soundness of lifted annotative evaluation factory) . . . . . . . . . 61
3 Theorem (Soundness of expression evaluation using $\hat{p}$) . . . . . . . . . . . 61
4 Theorem (Soundness of family-based analysis) . . . . . . . . . . . . . . . . 62
5 Theorem (Soundness of feature-product-based analysis) . . . . . . . . . . . 66
6 Lemma (Soundness of lifted compositional evaluation factory) . . . . . . . 68
6 Theorem (Soundness of expression evaluation using $\varphi$) . . . . . . . . . . . 68
7 Theorem (Soundness of feature-family-based analysis) . . . . . . . . . . . . 69
7 Lemma (r-equivalence for `ITE`) . . . . . . . . . . . . . . . . . . . . . . . . 71
8 Theorem (r-equivalence of variability encoding and derivation by composition) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
8 Lemma (Extensional equality for expression `ITE`) . . . . . . . . . . . . . . 74
9 Theorem (Soundness of variability encoding for expressions) . . . . . . . . 75
10 Theorem (Soundness of feature-family-product-based analysis) . . . . . . . 77

9 Lemma (Existence of minimal PMCs) . . . . . . . . . . . . . . . . . . . . . 162
10 Lemma (Existence of maximal PMCs) . . . . . . . . . . . . . . . . . . . . 162
11 Lemma (Derivation by composition terminates) . . . . . . . . . . . . . . . 163
12 Lemma (Compositional evaluation terminates) . . . . . . . . . . . . . . . . 163
13 Lemma (r-equivalence of total composition and evaluation) . . . . . . . . . 165
1 Corollary (r-equivalence of total composition with DTMCs and evaluation) 168
5 Theorem (Soundness of feature-product-based analysis) . . . . . . . . . . . 168
4 Lemma (Soundness of expression lifting) . . . . . . . . . . . . . . . . . . . 170
6 Lemma (Soundness of lifted compositional evaluation factory) . . . . . . . 171

7 Lemma (r-equivalence for `ITE`) . . . . . . . . . . . . . . . . . . . . . . . . 174
8 Theorem (r-equivalence of variability encoding and derivation by composition) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 176
8 Lemma (Extensional equality for expression `ITE`) . . . . . . . . . . . . . . 177
9 Theorem (Soundness of variability encoding for expressions) . . . . . . . . 178

# Acronyms

**ADD** Algebraic Decision Diagram.

**CTL** Computation Tree Logic.

**CTMC** Continuous-Time Markov Chain.

**DTMC** Discrete-Time Markov Chain.

**JML** Java Modeling Language.

**MDP** Markov Decision Process.

**PCTL** Probabilistic Computation Tree Logic.

**PMC** Parametric Markov Chain.

**PVS** Prototype Verification System.

**SPL** Software Product Line.

**TCC** Type-correctness Condition.

**UML** Unified Modeling Language.

# Contents

**List of Figures** **xiii**

**List of Tables** **xvi**

**List of Definitions** **xvii**

**List of Theorems** **xix**

**Acronyms** **xxi**

**1 Introduction** **1**
1.1 Problem Statement . . . . . . . . . . 2
1.2 Solution . . . . . . . . . . 3
1.3 Summary of Contributions . . . . . . . . . . 5
1.4 Outline . . . . . . . . . . 6

**2 Background** **9**
2.1 Software Product Lines . . . . . . . . . . 9
2.1.1 Main Concepts . . . . . . . . . . 10
2.1.2 Variability Implementation . . . . . . . . . . 13
2.1.3 Product-Line Analysis . . . . . . . . . . 15
2.2 Reliability Analysis . . . . . . . . . . 17
2.2.1 Parametric Markov Chains . . . . . . . . . . 20
2.2.2 Parametric Probabilistic Reachability . . . . . . . . . . 22
2.3 Algebraic Decision Diagrams . . . . . . . . . . 24
2.4 PVS . . . . . . . . . . 27

**3 Commuting Strategies for Product-line Reliability Analysis** **35**
3.1 DTMC Models of Product Lines . . . . . . . . . . 35
3.1.1 Annotative Models . . . . . . . . . . 38
3.1.2 Compositional Models . . . . . . . . . . 42

3.2 Reliability Analysis Strategies . . . 51
3.2.1 Product-based Strategies . . . 53
3.2.2 Family-based Strategies . . . 54
3.2.3 Feature-based Strategies . . . 63
3.2.4 Bridging Compositional and Annotative Models . . . 70
3.2.5 Feature-family-product-based Strategy . . . 76
3.3 Concluding Remarks . . . 78

**4 Formalization in PVS** **81**
4.1 Specification Strategy . . . 82
4.2 Walk-through . . . 85
4.2.1 Foundations . . . 85
4.2.2 Family-product-based Strategy . . . 94
4.2.3 Family-based Strategy . . . 97
4.2.4 Feature-based Strategies . . . 100
4.2.5 Variability-encoding . . . 106
4.3 Mechanization Effort . . . 115
4.3.1 Distribution of Lemmas . . . 116
4.3.2 Origin of Lemmas . . . 118
4.3.3 Proof Automation . . . 119
4.3.4 Theory Evolution . . . 124
4.4 Lessons Learned . . . 125
4.5 Limitations and Threats to Validity . . . 130
4.5.1 Axioms . . . 130
4.5.2 Unfinished Mechanized Proofs . . . 132

**5 Conclusions** **139**
5.1 Discussion of Results . . . 140
5.2 Threats to Validity . . . 141
5.3 Related Work . . . 143
5.4 Future Work . . . 148

**Bibliography** **151**

**Acronyms** **161**

**A Additional Proofs** **162**
A.1 Existence of Minimal and Maximal PMCs . . . 162
A.2 Termination Lemmas . . . 163

A.3 Soundness of Feature-product-based Analysis . . . . . . . . . . . . . . . . . 163
A.4 Lifting Lemmas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 170
A.5 Variability Encoding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173
A.5.1 Variability Encoding of PMCs . . . . . . . . . . . . . . . . . . . . . 173
A.5.2 Variability Encoding of Expressions . . . . . . . . . . . . . . . . . . 177

**B Probabilistic Models 180**

**C Mechanization Mapping 183**

**D Theory Dependencies 200**

# Chapter 1

# Introduction

Software product line engineering is a means to systematically manage variability and commonality in software systems, enabling the automated synthesis (*derivation*) of related programs (known as *variants* or simply *products*) from a set of reusable assets (known as *domain artifacts*) [3, 25, 73]. In a product line, variability is modeled in terms of features, which are distinguishable characteristics that are relevant to stakeholders of the system [26]. This methodology improves productivity and time-to-market, and it eases mass customization of software [73].

In recent years, product lines have been widely applied in both industry [8, 10, 45, 87, 89, 92, 94] and academia [3, 25, 35, 44, 73, 95], in particular to safety- and mission-critical systems [32, 33, 53, 76, 94]. Model checking is of particular interest to quality assurance of such systems. It is a verification technique that explores all possible system states in a systematic manner, effectively checking that a given system model satisfies a certain property [7]. Among the existing quality properties, this work focuses on *user-oriented reliability*, which is informally defined as the probability that the system will give the correct output in response to a typical set of input data [18].

However, the number of products in a product line may grow exponentially with the number of features, giving rise to an *exponential blowup* of the configuration space [3, 12, 22, 23]. The Linux kernel, for instance, has approximately 10,000 configuration options (i.e., features) [3]. Indeed, even smaller product lines, with tens to hundreds of features, may have millions of possible configurations [54], so it is often infeasible to quality-check each of these products in isolation. Nonetheless, product lines of safety-critical software (e.g., in the domains of avionics and medical systems) need to ensure that its products are reliable.

Since software verification techniques for the single-product case are widely used by the industry, it is beneficial to exploit their maturity to increase quality while reducing cost and risk [7]. Accordingly, a number of approaches to product-line analysis adapt

established analysis techniques—e.g., type checking, data-flow analysis, control-flow analysis, and theorem proving—to cope with variability [85]. In particular, several model checking techniques have been successfully lifted to operate on product lines [19, 21, 23, 24, 34, 51, 82, 85], some of which explicitly consider reliability as a probabilistic property [36, 54, 64, 76].

Such product-line analyses can be classified along three dimensions: product-based (the analysis is performed on generated products or models thereof), family-based (only domain artifacts and valid combinations thereof are checked), and feature-based (domain artifacts implementing a given feature are analyzed in isolation, regardless of their valid combinations) [85]. More than one dimension can be exploited in a given technique, giving rise to feature-family-based analyses (features are partially analyzed in isolation and then the intermediate results are combined in a family-based fashion) and family-product-based analyses (domain artifacts are partially analyzed considering only valid configurations, yielding a result that is prone to enumerative analysis), for instance.

These combined strategies have advantages and disadvantages, but the compromises are specific to each technique. Thus, empirical studies assess the actual trade-offs of different strategies in concrete usage scenarios [49, 54, 59, 92]. Moreover, it is still an open research question to perform automated (or even systematic) lifting of standard analysis techniques to correctly operate on product lines. Therefore, the soundness of proposed techniques must be proved for each case.

## 1.1 Problem Statement

The existing approaches to product line analysis often lift standard (single-product) analysis techniques to work with product lines in a family-based fashion [14, 19, 34, 62, 82, 91]. Soundness of these approaches is then demonstrated by comparison to the corresponding product-based strategy. 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 are equivalent to one another (i.e., they compute the same reliability). The lack of such a theory hinders formal reasoning on the relationship between the dimensions and derived analyses.

Indeed, proving that an analysis method yields a correct result is a fundamental issue, especially for critical systems. For instance, Lanna et al. [54] proposed a feature-family-based strategy to user-oriented reliability analysis of product-lines. This approach was empirically assessed, and the results indicate that the proposed technique outperforms the existing ones with respect to both elapsed time and memory usage. Specifically for a safety-critical product line of body-sensor networks [70] (with 16 features and 298 possible

configurations), Lanna et al. [54] found that most existing strategies would not be feasible if more than 5 new features were added in future versions. However, as long as there is no evidence that the different examined strategies are mutually equivalent, empirical studies comparing them will have limited results.

Furthermore, this problem is also relevant in a broader context: is there a principle and possibly automated way to lift a given specification and analysis technique to product lines [85]? Answers to that question may bring the ability to derive product-line analysis techniques that are correct by construction. On the other hand, we believe that such a broad investigation could benefit from having a corpus of specific theories, each relating all three analysis dimensions with regard to a single analysis technique.

## 1.2 Solution

In this work, we constrain our investigation to the analysis strategies assessed by Lanna et al. [54]—that is, we examine the correctness of product-line analysis techniques that are based on Discrete-time Markov Chain (DTMC) models of *user-oriented* reliability analysis [18].[1] This way, we aim to tackle the issue of not having correctness proofs about a specific kind of analysis.

**Research Question**

In the context of user-oriented product-line reliability analysis specified by means of DTMC models, is it true that different analysis strategies yield equivalent results for any given product line?

This question was partially answered with our previous work [16], which formally related different strategies in the product-based and family-based dimensions of product-line analysis, providing analytical evidence that they commute. Moreover, the formalized strategies were implemented[2] as a product line of product-line analysis tools, called ReAna-SPL [54] (publicly available at `https://github.com/SPLMC/reana-spl`), which also extended the set of supported strategies with one feature-product-based and one feature-family-based. The latter work found *empirical* evidence that the strategies commute.

This work leverages the aforementioned results to also cover the *feature-based* dimension, reusing definitions and theorems as much as possible. The key to achieving this reuse

[1] Such models are not general enough to represent *any* given system, but their assumptions are reasonable for some product lines of interest (e.g, a body-sensor network) [18, 54].

[2] As of now, it is an open issue to formally relate the existing implementation to the mathematical specification. Nevertheless, such correspondence is outside the scope of this work.

was to mathematically characterize a compositional model as a finite set of annotative models, along with an associated structure denoted by a well-founded dependency relation (more details on this are presented in Section 3.1). Furthermore, we employ *Algebraic Decision Diagrams* [6] to encode the variability in algebraic expressions in a way that optimizes arithmetic operations.

Based on the product-line analysis taxonomy proposed by Thüm et al. [85], we now present a formalization of a total of seven approaches to reliability analysis of product lines: two product-based, a family-based, a family-product-based, a feature-family-based, a feature-product-based, and a feature-family-product-based. In particular, the latter of these is the first approach to combine all three dimensions in a single strategy.[3] This formalization, which covers all three product-line analysis dimensions, provides analytical evidence that the strategies yield the same result for any product line within our context. Thus, a practitioner can choose among the existing strategies the one that is more suitable to the product line at hand.

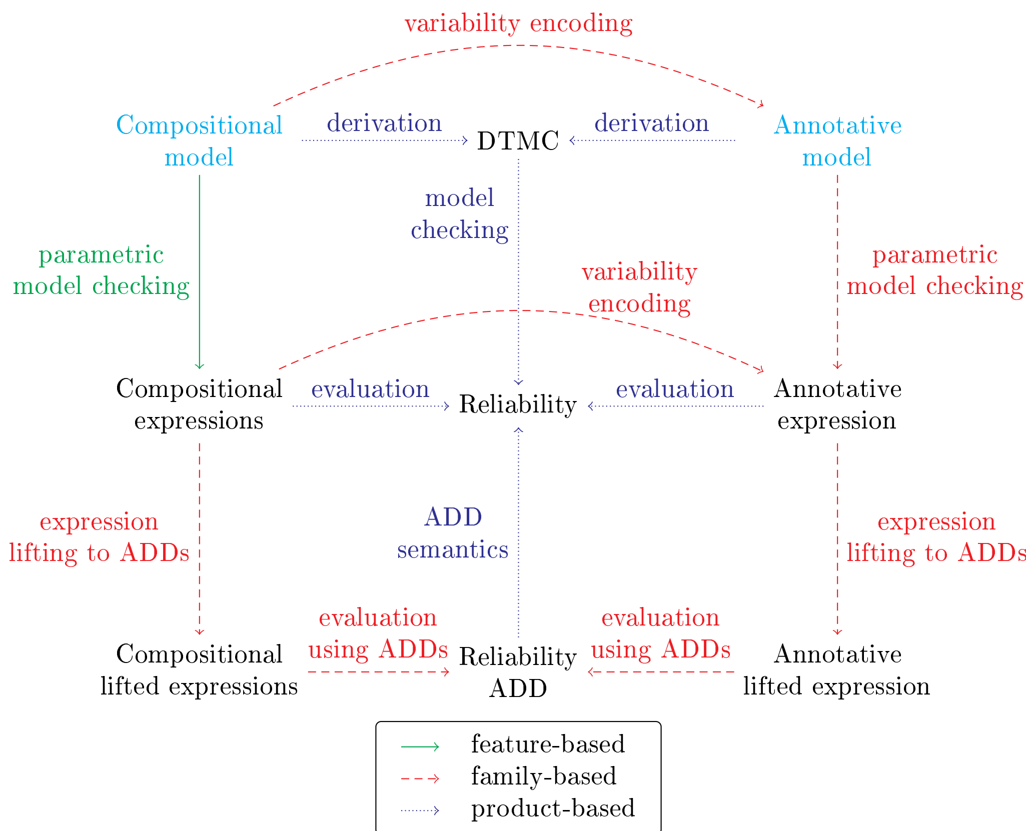



Figure 1.1: Overview of the commutative diagram of product-line reliability analysis strategies

[3]Thüm et al. [85] proposed that a feature-family-product-based analysis strategy would be possible, but the survey performed in their work did not find an instance of such strategy in the literature. To the best of our knowledge, no other work proposed a strategy in this category since then.

Figure 1.1 provides an overview of our commutativity results. We consider DTMC models of user-oriented software reliability with added support for either compositional or annotative variability [3] (upper left and upper right corners, respectively). Given a variability-enhanced reliability model of either kind, 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), covering all three product-line analysis dimensions. Thus, the arrows form an "analysis path" (a function composition) which defines the employed analysis strategy.

To further increase the evidence on the soundness of our commutativity theory, we mechanized our specification using the PVS proof assistant [67]. Lemmas and theorems were also specified and proved using PVS, which checks proofs and provides commands that automate some of their repetitive parts. This formalization of our theory in PVS allowed us to identify and correct some errors and imprecisions of the handcrafted version.

At the present moment, all elements (definitions, lemmas, and theorems) of the manual specification are fully mechanized, but the machine-verified proofs of 5 of the new auxiliary lemmas are still unfinished. Their correctness is manually argued, however.

We also present a report on the mechanization process, with the twofold purpose of getting feedback on the resulting PVS specification and sharing acquired knowledge with fellow researchers. Moreover, we believe that this report may contribute to future empirical studies over refactoring patterns and best practices regarding PVS specification and proofs.

## 1.3 Summary of Contributions

This work formally specifies a theory of reliability analysis of software product lines. Although our specifications have been developed to model an existing implementation [54], the contributions presented here are mainly analytical, abstracting implementation details and empirical assessment. Nonetheless, the core contribution of this work is a machine-verified proof that analysis strategies which have been empirically compared [54] are indeed sound.

We present the following peer-reviewed results [15]:

1. The formalization of seven strategies for reliability analysis of software product lines, covering all three analysis dimensions in the classification by Thüm et al. [85] (Section 3.2).

2. A novel feature-family-product-based strategy for model checking of product lines (Section 3.2.5). To the best of our knowledge, this is the first technique to combine all three product-line analysis dimensions.

3. Proofs of commutativity between different strategies (Section 3.2). This improves the current understanding on how analysis strategies for product lines relate to one another and establishes their soundness.

4. A commuting diagram of intermediate analysis steps (Figure 3.7), which relates different strategies and enables the reuse of soundness proofs between them.

5. A general principle for lifting analyses to product lines using algebraic decision diagrams (Section 3.2.2, Theorem 2).

Furthermore, we provide the following unpublished contributions:

6. A mechanized specification of our analysis strategies (`https://github.com/thiagomael/rome-specs`), increasing the confidence on the soundness of the formalized theory.

7. A report on the process of mechanizing the theory in PVS (Chapter 4), with the goal of aiding researchers in the interactive theorem proving community.

Overall, the commuting diagram resulting from this work (see Figure 3.7 for a more detailed view) presents reliability analysis steps in a compositional manner at a conceptual level, showing how the different types of product-line analyses compose and inter-relate in that context. Such view allows the organization and structuring of facts (e.g., commutativity of intermediate analysis steps) in a concise and precise manner, facilitating the communication of ideas. This contributes to a more comprehensive understanding of underlying principles used in these strategies, which we envision could help other researchers to lift existing single-product analysis techniques to yet under-explored variability-aware approaches.

We expect that, in the long term, the aforementioned contributions will be useful to lay a framework for a general theory of product-line analysis. Thus, this work indirectly contributes to the broader problem of lifting software analysis techniques to product lines.

## 1.4 Outline

This work is organized as follows:

- Chapter 2 presents fundamental concepts that are necessary for the discussion. It introduces software product lines and the corresponding analysis taxonomy, as well as the parametric behavioral models and decision diagrams leveraged by our analysis techniques. Additionally, it gives an overview of the PVS verification system [67].

- Chapter 3 corresponds to the published results of our research [15]. It presents our formalization of behavioral models for software product lines (Section 3.1), our analysis strategies (Section 3.2), and a formulation of the soundness of these strategies as theorems, along with corresponding proofs. To better illustrate the formal concepts, we also provide a running example.

- Chapter 4 shows the most relevant aspects of the mechanized version of our theory. This chapter also presents a discussion about the lessons learned in the process of PVS specification and computer-aided theorem proving. Last, we argue about threats to the validity of our mechanized theory.

- Chapter 5 discusses our conclusions and threats to their validity, along with related and future work.

- Appendix A presents the details of proofs that were summarized to improve the readability of Chapter 3.

- Appendix B contains the probabilistic models used in our running example in their entirety.

- Appendix C presents the correspondence between each element in the manual specification (Chapter 3) and its mechanized counterpart (Chapter 4). This appendix also presents a description of our PVS theories and a diagram depicting the dependencies between them.

- Appendix D is a compilation of dependency graphs for the main theorems presented in this work. These diagrams have been used throughout our research to assess the impact of changes, but they are also useful to visualize the relationship between the elements in our theory.

Table 1.1 relates contributions to their corresponding location within this work.

Table 1.1: Research outline

| **Contribution** | **Location** |
|---|---|
| Item 1 | Chapter 3 |
| Item 2 | Section 3.2.5 |
| Item 3 | Section 3.2 |
| Item 4 | Figure 3.7 |
| Item 5 | Section 3.2.2 |
| Item 6 | Chapter 4 |
| Item 7 | Chapter 4 |

# 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 2.1.3). Within this domain, this work focuses on user-oriented 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. Last, we provide an overview of PVS, the interactive theorem prover used to create the machine-verified version of our theory (Section 2.4).

## 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 systems. 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 [3], 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 bug-fix 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* [31]. 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 [80]. 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,[1] 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 [25]. 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 [3, 73, 87].

The main concern in product-line engineering is managing variability, which is defined by van Gurp et al. [88] 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 [3]. 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,

[1] https://www.kernel.org/doc/Documentation/kbuild/

reuse, and variation across all phases of the software life cycle [3]. The features of a product line and their relationships are documented in a *feature model* [26, 46], 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. We also restrict our scope to finite feature sets.

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 formulas. As an example, one can state $\mathtt{f} \Rightarrow \mathtt{g}$, 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 [76]. 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 configuration.

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 formulas 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 atomic propositions

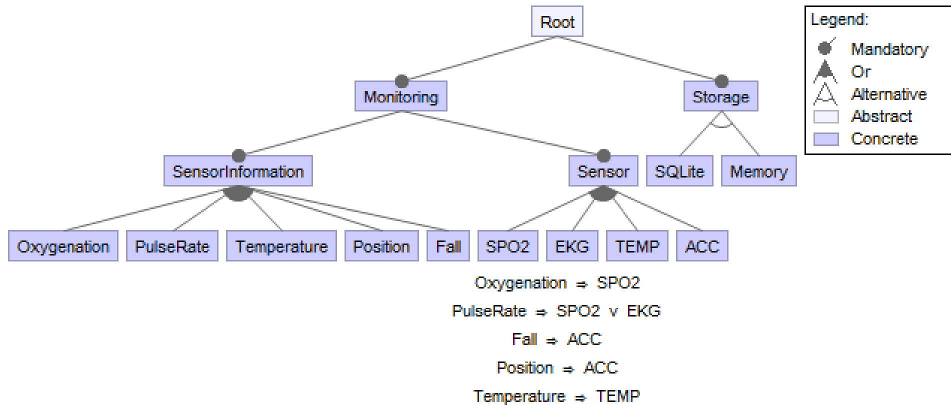



Figure 2.1: Feature model of the BSN product line [76]

regarding features, 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 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* [26]. 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* [3]. Actual behavior is included or excluded from a generated product by means of *presence conditions*, which are propositional formulas over features [27]. 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. Therefore, we define an arbitrary (but fixed) total order of features by turning the set $F$ of features into a list. This way, we can unambiguously 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, with the $i$-th position denoting presence or absence of the $i$-th feature[2] by the values 1 and 0, respectively. 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 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 [3]:

**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.

[2] The actual order of features does not affect our results, since its only purpose is to consistently refer to values in Boolean tuples.

*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* [39, 86], 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.

A usual annotative technique is the use of preprocessor directives, which is the variability representation mechanism in the Linux Kernel [69]. 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 [48].** 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 [9, 75].** 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* [78] is a well-known example of transformation-based (or *transformational*) approach [38]. 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 [1] and business processes [58]. Teixeira et al. [81] 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) [67]. This work, in particular, uses both annotative and compositional approaches to handle variability in probabilistic models of product lines.

### 2.1.3 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 [3]. 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*.

There is a number of approaches to product-line analysis that adapt established analysis techniques—e.g., type checking, data-flow analysis, control-flow analysis, and theorem proving—to cope with variability [85]. In particular, several model checking techniques have been successfully lifted to operate on product lines [19, 21, 23, 24, 34, 36, 51, 64, 76, 82, 85].

Thüm et al. [85] performed a survey on analysis strategies for 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* [71]. 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 [7]. 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 [67], or can be generated from invariant specifications declared in the source code using the Java Modeling Language (JML) [55], for instance.

Among the studies regarding the application of these techniques to product-line analyses, Thüm et al. [85] categorized three dimensions of analysis strategies for product lines:

**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 [4].

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. [83] propose formal verification of design-by-contract properties [61] 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*.

Strategies that combine different analysis dimensions are classified as follows [85]:

**Feature-product-based.** Consists of a feature-based analysis followed by a product-based analysis. This strategy leverages the feature-based phase to ease the analysis effort necessary for the enumerative phase.

**Feature-family-based.** In this strategy, one performs a feature-based analysis to check properties that apply individually for each feature, then the results are combined to undergo a family-based analysis. This last phase considers the feature model constraints and the interactions between features all at once, enabling the analysis of properties that are not observable in the scope of a single feature.

**Family-product-based.** This strategy consists of a partial family-based analysis followed by a product-based analysis that leverages the intermediate results. Such an approach is useful when the available resources are not sufficient for a complete family-based analysis, for instance.

**Feature-family-product-based.** In this strategy, we perform a feature-based analysis followed by a family-product-based analysis that leverages the analysis effort of the feature-based phase. According to the survey by Thüm et al. [85], there are no concrete instances of this strategy in the literature.

Those different analysis strategies have been applied in the context of different analysis techniques [85]. However, the trade-offs involved cannot be inferred for the general case. Empirical studies have to be performed to assess the actual advantages and disadvantages of different strategies in concrete usage scenarios [49, 54, 59, 92].

## 2.2 Reliability Analysis

The theory of reliability analysis strategies developed in this work formalizes and extends the techniques presented by Lanna et al. [54]. Hence, we follow the same approach of considering software reliability from a user's perspective.

Such *user-oriented reliability* of a software program in a given user environment is defined as the probability that the program will give the correct output with a typical set

of input data from that user environment [18]. Accordingly, we model software behavior in a state-space-based fashion, by means of a Discrete-time Markov Chain (DTMC) in which states represent (parts of) software modules and transitions represent either a possible transfer of control between modules (with an associated probability) or a module execution failure (with probability $1 - r$, where $r$ is the module reliability). We assume that module reliabilities are independent from one another and that the transfer of control between modules depends only on the module currently executing (i.e., transfer of control is a Markov process). Moreover, we constrain this model to have a single initial state (representing the program entry point) and only two terminal (absorbing) states, representing program success (i.e., correct execution) and program failure.

A model of software reliability built this way follows the principles presented by Cheung [18]. Additionally, this type of model encodes the following assumptions of the work by Lanna et al. [54]:

- Model states represent the execution of a function by some system component, as described in UML behavioral diagrams [66]. As such, these states abstract actual program states (e.g., variable values) and can be contained within a finite set.
- Reliability models are *time-homogeneous*, meaning that each of the transition probabilities is constant over time.
- As a scope limitation, we do not model parallelism or nondeterminism.

Constructing models with the aforementioned constraints, we view the reliability of a system as the probability that, starting from the initial state, the system eventually reaches the success state [18]. This reliability property is then computed as a *reachability probability* in the DTMC that serves as the reliability model.

To perform this computation, we define a DTMC as 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 a (possibly singleton) set of target states that are to be reached as a success measure.[3] Moreover, each row of the transition probability matrix sums to 1, that is, $\forall_{s \in S} \cdot \mathbf{P}(s, S) = 1$, where $\mathbf{P}(s, S) = \sum_{s' \in S} \mathbf{P}(s, s')$.

For every state $s \in S$, we say that a state $s'$ is a *successor* of $s$ iff $\mathbf{P}(s, s') > 0$. Accordingly, the set of successor states of $s$, $Succ(s)$, is defined as $Succ(s) = \{s' \in S \mid \mathbf{P}(s, s') > 0\}$. A DTMC induces an underlying digraph where states act as vertices and edges link states to their successors. 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

[3] This definition departs from the one by Baier and Katoen [7] in two ways: (a) we abstract the possibility of multiple initial states and the computation of other temporal properties (to focus on reliability analysis) and (b) we incorporate target states in the model (to abbreviate model checking notation).

$p = \mathbf{P}(s, s')$. This way, we say that a state $s'$ of a DTMC is *reachable* from a state $s$, denoted by $s \rightsquigarrow s'$, iff $s'$ is reachable from $s$ in the DTMC's underlying digraph. Likewise, we write $s \not\rightsquigarrow s'$ to denote that $s'$ is unreachable from $s$. This notation is also used with respect to a set $T$ of states: $s \rightsquigarrow T$ iff there is at least one state $s' \in T$ such that $s \rightsquigarrow s'$, and $s \not\rightsquigarrow T$ otherwise.

Given a DTMC $\mathcal{D} = (S, s_0, \mathbf{P}, T)$, a state $s \in S$, and a set $T \subseteq S$ of target states, the probability of reaching a state $t \in T$ starting from $s$ (within any number of transitions) is denoted by $Pr^{\mathcal{D}}(s, T)$. Whenever $T$ is a singleton set whose only member is a state $t$, we write $Pr^{\mathcal{D}}(s, t)$ for brevity.

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).

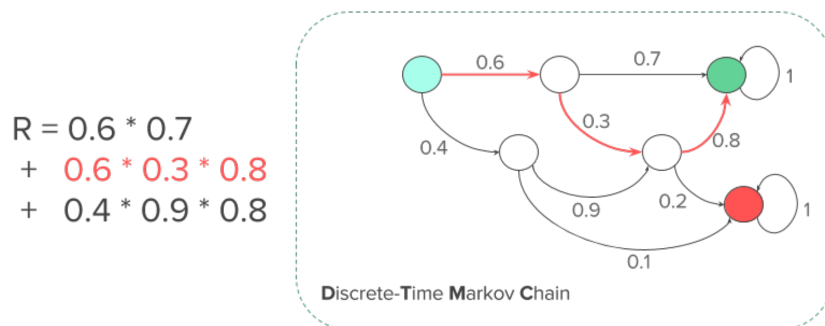



Figure 2.2: Example graph view of a DTMC and the corresponding reachability probability

Reliability analysis in our setting can be summarized as the process through which we determine the probability $p$ for which the formula $P_{=p}[\Diamond success]$ holds, where *success* is a proposition that only holds true for $s \in T$. This formula is specified using Probabilistic Computation Tree Logic (PCTL) [42] and states that $p$ is the probability that *success* will eventually hold. This property follows the *probabilistic existence* pattern of probabilistic properties specification, which is one recognized way of specifying reliability [37]. Other possible views of reliability property include the probability that a system does not fail within a given time interval, which can be specified using *probabilistic invariance*, for instance. Nevertheless, handling those alternative views is out of scope.

The reachability probability for a DTMC can be computed using probabilistic model checking algorithms, implemented by off-the-shelf tools such as PRISM [52] and PARAM [40]. 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 [7], that suits the purpose of this work.

**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 $T \subseteq S$ of target states, the probability of reaching a state $t \in T$ from $s$ satisfies the following property:

$$Pr^{\mathcal{D}}(s, T) = \begin{cases} 1 & \text{if } s \in T \\ 0 & \text{if } s \not\rightsquigarrow T \\ \sum_{s' \in S} \mathbf{P}(s, s') \cdot Pr^{\mathcal{D}}(s', T) & \text{if } s \notin T \wedge s \rightsquigarrow T \end{cases}$$

In a product line, different products give rise to distinct behavioral models. To handle the behavioral variability that is inherent to product lines, we resort to *Parametric Markov Chains* [29].

### 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 [19, 36, 76].

**Definition 1** (Parametric Markov Chain)**.** A Parametric Markov Chain is defined by Hahn et al. [41] 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 $\varepsilon$ 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 $u : X \to \mathbb{R}$ 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 $\varepsilon$ by $u(x)$, also denoted by $\varepsilon[x_1/u(x_1), \ldots, x_n/u(x_n)]$.

For instance, suppose we have sets of variables $X = \{x, y\}$ and $X' = \{x, y, z\}$, and an evaluation $u = \{x \mapsto 2, y \mapsto 5, z \mapsto 3\}$. If $\varepsilon \in \mathcal{F}_X$ is the rational expression $x - 2y$, then $\varepsilon[X/u] = \varepsilon[x/2, y/5] = 2 - 2 \cdot 5 = -8$. Note that, if $u$'s domain, $X'$, is different from the set $X$ of variables in $\varepsilon$, then $\varepsilon[X/u] = \varepsilon[(X \cap X')/u]$.

**Definition 2** (Expression evaluation)**.** Given expressions $\varepsilon_1$ and $\varepsilon_2$ 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)
- $\mathbf{P}_u(s, S) = 1$ (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 evaluations.[4] 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 $\varepsilon$ 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).

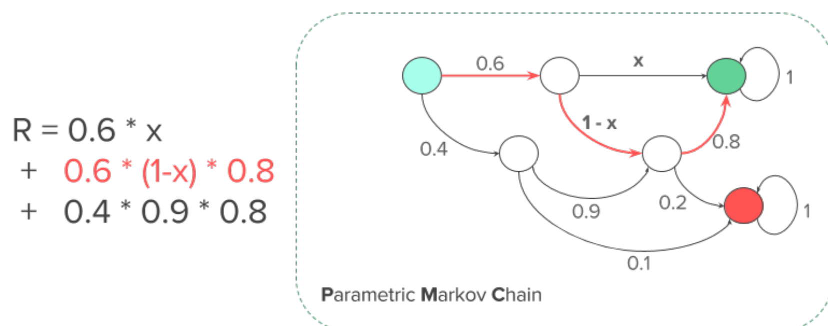



Figure 2.3: Example graph view of a PMC and the intuition for the corresponding reachability probability expression

Hahn et al. [41] present a parametric probabilistic reachability algorithm (Algorithm 1) and prove that evaluating the resulting expression $\varepsilon$ 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.

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

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

[4]Hahn et al. [41] actually define an evaluation in a more general way as a *partial* function. However, since we only deal with well-defined evaluations (which are total by definition [41]), we are able to simplify the definitions in this work by using total functions.

**Algorithm 1** Parametric Reachability Probability for PMCs [41]

**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** $(s_{pre}, s_{succ}) \in \mathit{Pre}(s) \times \mathit{Succ}(s)$ **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** $\frac{1}{1-\mathbf{P}(s_0,s_0)}\mathbf{P}(s_0, T)$

Once this computation has been performed for all predecessor ($\mathit{Pre}(s)$) and successor states ($\mathit{Succ}(s)$), $s$ itself is *eliminated* from the set $S$ of states, and the process starts again by arbitrarily picking another state.

Figure 2.4 [41] 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 $\sum_{i=0}^{\infty} p_a (p_c)^i p_b = p_a (\sum_{i=0}^{\infty} p_c^i) p_b = p_a \frac{1}{1-p_c} p_b$.[5]

**Definition 4** (State elimination step)**.** Given a PMC $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$ and an arbitrary state $s \in S$, a state elimination step of the algorithm by Hahn et al. [41] updates the transition matrix $\mathbf{P}$ to $\mathbf{P}'$, such that, for all states $s_{pre}, s_{succ} \in S \setminus \{s\}$,

$$\mathbf{P}'(s_{pre}, s_{succ}) = \mathbf{P}(s_{pre}, s_{succ}) + \mathbf{P}(s_{pre}, s) \cdot \frac{1}{1-\mathbf{P}(s,s)} \cdot \mathbf{P}(s, s_{succ})$$

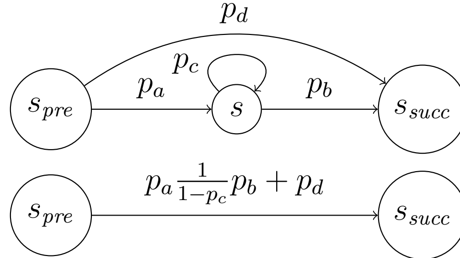



Figure 2.4: Elimination of state $s$ in the parametric reachability probability algorithm (adapted from Hahn et al. [41])

[5] Whenever $0 < x < 1$, we have the following convergent sum: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$.

The soundness of the parametric probabilistic reachability algorithm by Hahn et al. [41] is expressed by the following lemma and summarized by the commuting diagram in Figure 2.5.

**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* $\varepsilon$ *be the output of the parametric probabilistic reachability algorithm by Hahn et al. [41] (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. [41] 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. [41] for more details on the algorithm itself and the proof mechanics. □

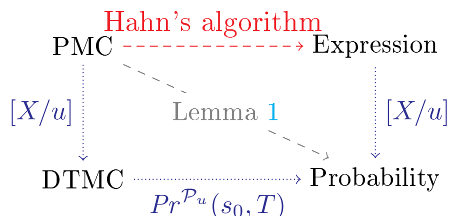



Figure 2.5: Statement of Lemma 1

## 2.3 Algebraic Decision Diagrams

Thus far, we have introduced software product lines and the parametric model checking technique that we employ to compute the reliability of a product line as a function of its configuration options—denoted by a rational expression. However, even though rational expressions are conceptually simpler than Markov models, evaluating an expression for every possible configuration may still be infeasible for certain product lines. Hence, we introduce ADDs as enablers of non-enumerative evaluation.

An Algebraic Decision Diagram (ADD) [6] is a data structure that encodes $k$-ary Boolean functions $\mathbb{B}^k \rightarrow \mathbb{R}$. As an example, Figure 2.6 depicts an ADD representing the following binary function $f$:

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

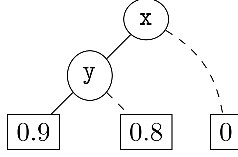



Figure 2.6: ADD $A_f$ representing the Boolean function $f$ in Equation (2.2)

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).

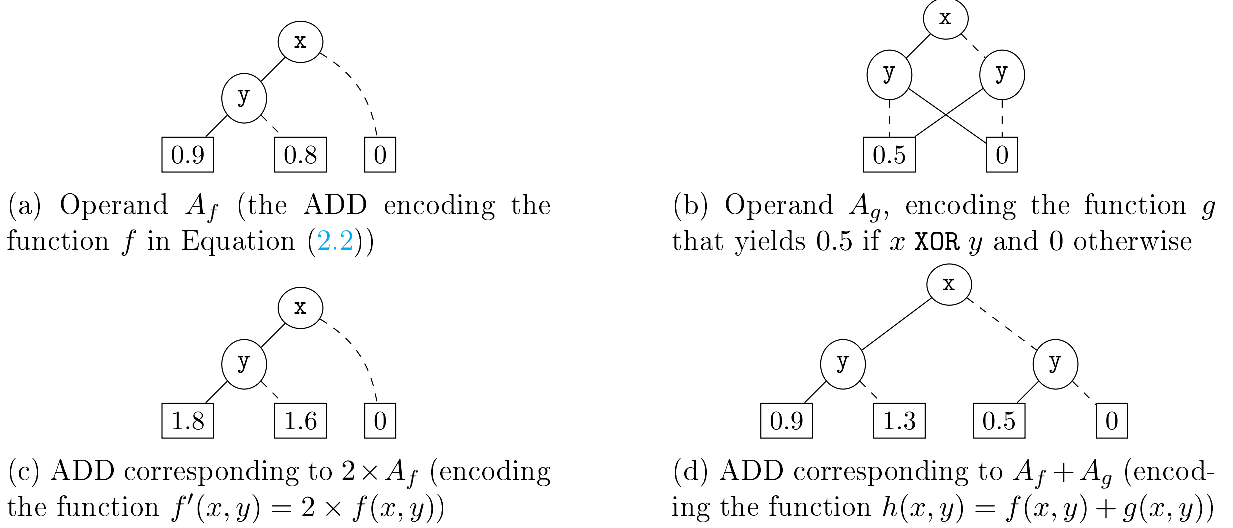
In the example, to evaluate $f(1, 0)$, we start in the `x` node, take the solid edge to node `y` (since the actual parameter $x$ is 1), then take the dashed edge to the terminal 0.8. Thus, $f(1, 0) = 0.8$. Henceforth, we will use a function application notation for ADDs, meaning that, if $A$ is an ADD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

ADDs have several applications, two of which are of direct interest to this work. The first one is the efficient application of arithmetics over Boolean functions. We employ Boolean functions to represent mappings from product-line configurations (Boolean tuples) to their respective reliabilities. An important aspect that motivated the use of ADDs for this variability-aware arithmetics is that the enumeration of all configurations to perform Real arithmetics on the corresponding reliabilities is usually subject to exponential blowup. ADD arithmetic operations are linear in the input size, which, in turn, can also be exponential in the number of Boolean parameters (i.e., ADD variables), in the worst case. However, given a suitable variable ordering, ADD sizes are often polynomial, or even linear [6]. Thus, for most practical cases, ADD operations are more efficient than enumeration.

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 2.7, we see two examples of ADD arithmetics. The first and simpler one (Figure 2.7c) shows the multiplication of the ADD $A_f$ (Figure 2.7a) by the constant factor 2. This operation takes place by multiplying terminals by the given factor. The second example (Figure 2.7d) shows the sum of ADDs $A_f$ and $A_g$ (Figure 2.7b), 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.

As previously mentioned, 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 $x$ `XOR` $y$ 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 2.7: 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. $\forall_{\odot \in \{+,-,\times,\div\}} \cdot (A_1 \odot A_2)(\bar{b}) = A_1(\bar{b}) \odot A_2(\bar{b})$

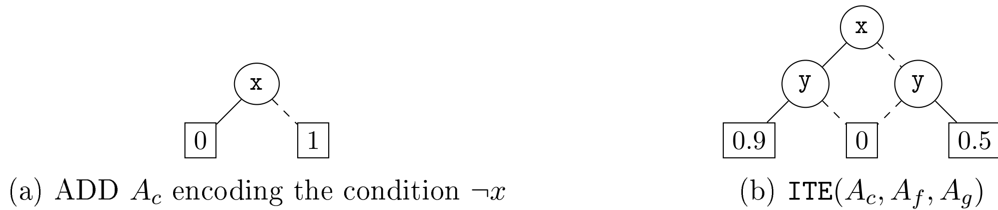
2. $\forall_{i \in \mathbb{N}} \cdot A_1^i(\bar{b}) = A_1(\bar{b})^i$

The second application 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 2.8. This figure depicts an ADD resulting from $\texttt{ITE}(A_c, A_f, A_g)$ (Figure 2.8b), where $A_c$ (Figure 2.8a) encodes the function $c(x, y) = \neg x$, and the ADDs $A_f$ and $A_g$ are taken from Figures 2.7a and 2.7b. As with ADD arithmetics, the details of the ADD `ITE` operation are omitted for being out of scope.

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




(a) ADD $A_c$ encoding the condition $\neg x$ (b) $\texttt{ITE}(A_c, A_f, A_g)$

Figure 2.8: Example of an `ITE` operation over ADDs

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 2.9. This ADD encodes the same function $f$ (Equation (2.2)) as the ADD $A_f$ in Figure 2.6, 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.

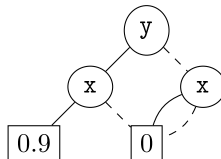



Figure 2.9: Alternative ordering for encoding the Boolean function $f$ in Equation (2.2) as an ADD

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 2.7b 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. [6].

## 2.4 PVS

To increase confidence in the theory developed in this work, we developed a machine-verified theory using the PVS interactive theorem prover. Thus, we provide an overview of some language constructs in PVS 6.0 that are used throughout our specification. However, this section is not a complete tutorial or reference guide to all features of PVS; for detailed instructions on how to develop and verify a mechanized specification using this tool, the reader is referred to the official documentation [67, 68, 79].[6]

[6] http://pvs.csl.sri.com/documentation.shtml

The Prototype Verification System (PVS) is a set of integrated tools that assist the development and verification of formal specifications [68]. PVS theories are defined using a specification language [67] that supports expressive features such as higher-order functions and dependent types. Moreover, this language allows the definition of properties about the specified functions and types, in the form of theorems or axioms. To verify that theories written in the specification language are semantically consistent, PVS also provides a type checker and an interactive theorem prover [79].

Whenever the type checker is not able to decide whether a given definition is consistent or not, it generates a *type-correctness condition* (TCC)—a proof obligation that the user is required to discharge. Both TCCs and user-defined theorems can be proved interactively with PVS's theorem prover. Although PVS provides several strategies to automate proof steps, its focus is not on *automatic* proving, but rather on *interactive proof checking*.

A specification in PVS is a collection of theories, and a theory is composed of definitions of types and functions, plus the associated theorems and axioms. A type can be interpreted (i.e., defined in terms of other types) or uninterpreted. For instance, we can state that there exists a set of variables by using `variable: TYPE+`, whereby the only assumptions made are that this type is nonempty and disjoint from other types (except its own subtypes). An interpreted type, on the other hand, is defined in terms of other types:

```
1 evaluation: TYPE = [variable -> real]       % function type
2 complex: TYPE = [# r: real, im: real #]     % record type
3 nzreal: TYPE = {x: real | x /= 0}           % ℝ*
```

The latter example defines the type of non-zero Reals, $\mathbb{R}^*$, using a notation that closely resembles set comprehension. This PVS construct is called a *predicate subtype*, since it defines a subtype of `real` (the *base* type) in terms of a predicate. Alternatively, this definition could be given as follows:

```
1 nonzero?(x: real): boolean = x /= 0 % predicate for x ≠ 0
2 nzreal: TYPE = (nonzero?)         % syntactic sugar for {x: real | nonzero?(x)}
```

An interesting feature of PVS is that its standard library, called *Prelude*, represents a set of elements of type `T` (`set[T]`) as a predicates of type `[T -> boolean]`. This way, set membership is tested using function application, and we can define types from set elements using the predicate subtyping mechanism:

```
1 X: set[variable]     % a constant declaration
2 v: (X)               % another constant, of type {v: variable | X(v)}
```

This facility is extensively used throughout our mechanized specification to declare *dependently-typed* function parameters.

```
1 eval(X: finite_set[variable], u: [(X) -> real]): finite_set[real]
```

The above line presents a higher-order function that takes two parameters: a finite set `X` of variables and a function `u` that maps *elements* of `X` to Real numbers. The type of `u` is dependent on the *value* of X; in practice, `u` is defined as a partial function in the domain of `variables`.

Moreover, `eval` in this example is an *uninterpreted* function (i.e., an uninterpreted constant of a function type). To specify the semantics of this function, one can either use an axiom or add a function definition (i.e., turn it into an interpreted function constant):

```
% Axiomatic style
eval(X: finite_set[variable], u: [(X) -> real]): finite_set[real]
eval_semantics: AXIOM
    FORALL (X: finite_set[variable], u: [(X) -> real]):
        eval(X, u) = {r: real | EXISTS (x: (X)): u(x) = r}

% Alternative using definition
eval_def(X: finite_set[variable], u: [(X) -> real]): finite_set[real] =
    {r: real | EXISTS (x: (X)): u(x) = r}
```

The main difference is that PVS guarantees that definitions preserve the consistency of a theory, whereas the user must manually verify that an axiom does not introduce inconsistencies [67]. In the previous example, for instance, PVS generates a TCC for `eval_def`, requiring us to prove that the set comprehension in Line 9 is indeed a finite set (because of the return type in Line 8). On the other hand, no proof obligation is created for the axiomatic version of the same specification, even though the return types (Lines 2 and 8) and the predicates (Lines 5 and 9) are the same for both specification alternatives. Since inconsistent specifications can be used to prove anything at all (i.e., they are useless for proving soundness of theories), it is best to avoid introducing axioms.

The definition of `eval_def` in the last example uses a *declarative* style, whereby we define a predicate that the function output must satisfy. An alternative is to use an *operational* style—i.e., to specify *how* the declared function produces that value. An operational alternative to the `eval` function is the following recursive definition:

```
eval_op(X: finite_set[variable], u: [(X) -> real])
        : RECURSIVE finite_set[real] =
    IF empty?(X)
      THEN emptyset
      ELSE add(u(choose(X)),
                eval_op(rest(X), restrict(u)))
    ENDIF
MEASURE card(X)
```

In this definition, we pick any element of `X` (function `choose` in Line 5), apply the function `u` to it, then add the result to set obtained by recursively applying `eval_op` to

the remaining elements of `X` (function `rest` in Line 6). The base case of this recursion occurs when `X` is empty (Line 4).

An interesting point to note is that PVS only allows the definition of terminating functions. Thus, we have to provide a measure (Line 8) that decreases with each call according to some well-founded relation. In this particular case, we use the cardinality of `X`, and the well-founded relation that establishes the notion of decreasing is inferred to be the "less-than" relation over the Natural numbers ($<$). The type checker then generates a TCC requiring us to prove that this measure is indeed decreasing for any input `X`:

```
eval_op_TCC3: OBLIGATION
  FORALL (X: finite_set[variable]):
      NOT empty?(X) IMPLIES card(rest(X)) < card(X)
```

The syntax for user-defined theorems follows the same pattern, but using other keywords—`THEOREM`, `LEMMA`, and `COROLLARY`.

PVS also supports the definition of abstract datatypes, such as the following type that we use to extended the Reals with an undefined value ($\mathbb{R} \cup \{\perp\}$).

```
maybe_real: DATATYPE
  BEGIN
      a_real(num: real): is_real?
      undefined: undefined?
  END maybe_real
```

Each line defines a *constructor*, a (possibly empty) set of *accessors*, and a *recognizer*—a predicate that is true for elements built with the corresponding constructor. In this particular case, an element `v` of the type `maybe_real` may be the constant `undefined`, or a value obtained by applying the `a_real` constructor to a Real number. If the `is_real?` recognizer returns true, the `num` accessor can be applied to obtain the encapsulated Real value. In PVS language, we can state this fact as the following (trivial) theorem:

```
trivial_fact: THEOREM
    FORALL (x: real): num(a_real(x)) = x
```

We can define arithmetic operations over `maybe_real` by overloading PVS operators already defined over `real` and specifying that any operation where at least one of the operands is undefined yields `undefined` as a result.

```
% Overloading of the sum operator
; +(a,b): maybe_real =
    IF (undefined?(a) OR undefined?(b))
        THEN undefined
        ELSE a + b
    ENDIF
```

Whenever the result of an operation over `maybe_real` is used in a context where a `real` is expected, PVS will generate an obligation to prove that this result satisfies the predicate represented by the `is_real?` recognizer. To avoid the generation of such TCCs when the operands are known to be `is_real?`, the PVS user can define *judgements* to make this property available to the type checker.

```
m, n: VAR (is_real?)     % logical variable to abbreviate declarations
sum_real: JUDGEMENT
    FORALL (m, n): (m + n) HAS_TYPE (is_real?)
sub_real: JUDGEMENT
    FORALL (m, n): (m - n) HAS_TYPE (is_real?)
mul_real: JUDGEMENT
    FORALL (m, n): (m * n) HAS_TYPE (is_real?)
```

PVS will require each judgement to be proved, by means of TCCs. To assist the design and verification of proofs for TCCs and user-defined theorems, PVS provides an interactive theorem prover [79]. This prover is based on the sequent calculus framework and supports a number of rules and proof strategies with different degrees of automation. For instance, there are rules for propositional simplification, quantifier instantiation, introduction of Skolem constants, induction, term rewriting, and simplification using decision procedures for equality and linear arithmetics (e.g., the highly-automated `grind`).

As an example (taken from the NASA PVS tutorial[7]), suppose we want to prove the following fact:

$$\forall x, y \in \mathbb{R}^+ \cdot x < y \implies x^2 < y^2$$

This theorem can be stated in PVS as follows:

```
squared_increasing: THEOREM
    FORALL (x,y:posreal):
        x < y IMPLIES x^2 < y^2
```

To prove that theorem, we start the interactive theorem prover by issuing the command `M-x prove`.[8] The prover starts with a sequent consisting of only the fact that we want to prove as a consequent:

```
squared_increasing :

  |-------
{1}   FORALL (x, y: posreal): x < y IMPLIES x ^ 2 < y ^ 2

```

[7] https://shemesh.larc.nasa.gov/PVSClass2012/pvsclass2012/index.html

[8] The combination `M-x` is achieved by pressing the `Alt` and `x` keys simultaneously

```
Rule?
```

We begin the proof by introducing Skolem constants for the universally quantified variables `x` and `y`, using the prover command `skeep` (Skolemize and keep the names of variables for the introduced constants).

```
Rule? (skeep)
Skolemizing and keeping names of the universal formula in (+ -),
this simplifies to:
squared_increasing :

{-1}  x < y
  |-------
{1}   x ^ 2 < y ^ 2

Rule?
```

Note that the implication was automatically flattened, so that the premise became an antecedent formula with index `[-1]`. Now, we expand the definition of the `^` operator.

```
Rule? (expand "^")
Expanding the definition of ^
this simplifies to:
squared_increasing :

[-1]  x < y
  |-------
{1}   expt(x, 2) < expt(y, 2)

Rule?
```

We see that `^` is actually syntactic sugar for the `expt` function. The next step is to leverage lemma `both_sides_expt_pos_lt_aux`, which is part of the *Prelude* built-in PVS library.

```
Rule? (lemma "both_sides_expt_pos_lt_aux")
Applying both_sides_expt_pos_lt_aux
this simplifies to:
squared_increasing :

{-1}  FORALL (m: nat, px, py: posreal):
        expt(px, m + 1) < expt(py, m + 1) IFF px < py
[-2]  x < y
  |-------
[1]   expt(x, 2) < expt(y, 2)

```

```
12 Rule?
```

The command `lemma` brings a lemma into the sequent as an antecedent formula with index `[-1]`. Note, however, that the formula `x < y`, whose index was previously `[-1]`, is now at the position `[-2]`.

The next proof step is to instantiate the lemma recently brought into the sequent:

```
1  Rule? (inst - "1" "x" "y")
2  Instantiating the top quantifier in - with the terms:
3  1, x, y,
4  this simplifies to:
5  squared_increasing :
6
7  {-1}  expt(x, 1 + 1) < expt(y, 1 + 1) IFF x < y
8  [-2]  x < y
9    |-------
10 [1]   expt(x, 2) < expt(y, 2)
11
12 Rule?
```

Now, the antecedent contains all the information that is needed to prove the consequent. Thus, issuing the command `assert` would apply decision procedures and finish the proof. However, we can also do it manually, by first simplifying the sum, then flattening the equivalence in formula `[-1]` to get two implications.

```
1  Rule? (simplify)
2  Simplifying with decision procedures,
3  this simplifies to:
4  squared_increasing :
5
6  {-1}  expt(x, 2) < expt(y, 2) IFF x < y
7  [-2]  x < y
8    |-------
9  [1]   expt(x, 2) < expt(y, 2)
10
11 Rule? (flatten)
12 Applying disjunctive simplification to flatten sequent,
13 this simplifies to:
14 squared_increasing :
15
16 {-1}  expt(x, 2) < expt(y, 2) IMPLIES x < y
17 {-2}  x < y IMPLIES expt(x, 2) < expt(y, 2)
18 [-3]  x < y
19   |-------
20 [1]   expt(x, 2) < expt(y, 2)
```

```
21
22 Rule?
```

Last, we use `prop` to apply propositional simplification (in this case, *modus ponens*) using the implication at `[-2]` and the fact at `[-3]`.

```
1  squared_increasing :
2
3  {-1}  expt(x, 2) < expt(y, 2) IMPLIES x < y
4  {-2}  x < y IMPLIES expt(x, 2) < expt(y, 2)
5  [-3]  x < y
6    |-------
7  [1]   expt(x, 2) < expt(y, 2)
8
9  Rule? (prop)
10 Applying propositional simplification,
11 Q.E.D.
```

The theorem prover also has the ability to store and replay proofs, and there are also facilities to report proof status and to perform proof chain analysis—i.e., check if all the lemmas appearing in a given proof have themselves been proved or stated as axioms or definitions, and if all TCCs have been discharged. Those capabilities are useful during the development of a mechanized theory, since complete proofs can be re-checked in response to changes in the specification.

# Chapter 3

# Commuting Strategies for Product-line Reliability Analysis

This chapter presents the formalization of our behavioral models for software product lines (Section 3.1) and of our analysis strategies (Section 3.2). It also presents a formulation of the soundness of our strategies as theorems, along with corresponding proofs. Last, we conclude with remarks on the applicability of our mathematical theory to other analysis strategies and to related domains (Section 3.3).

The contents presented hereafter are the purely mathematical results of the research, and correspond to the journal article *All roads lead to Rome: Commuting strategies for product-line reliability analysis* [15]. The discussion on annotative models (Section 3.1.1) and analyses thereof (Section 3.2.2), that is, the family-based and family-product-based strategies, is largely based on previous work [16]. Nonetheless, we include it here both for the sake of completeness and because some of the notation has evolved during peer review.

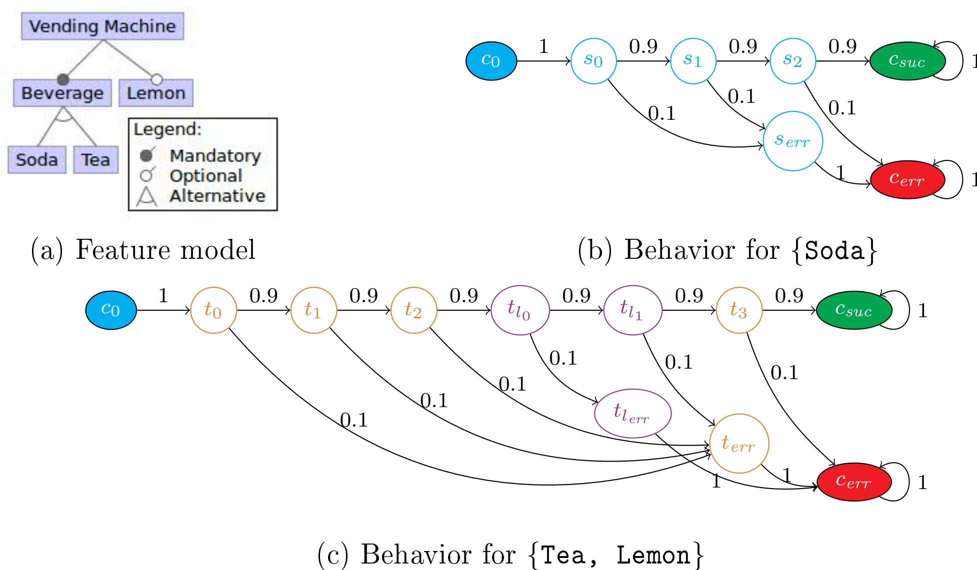
## 3.1 DTMC Models of Product Lines

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 3.2. 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 [36, 54, 65] or feature-oriented formalisms [19].

Since single-product analysis relies on DTMCs to model software behavior, we leverage the parameters in 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 3.1), slightly modified from the examples in the work by Ghezzi and Sharifloo [36] and Classen et al. [22] for didactic purposes. 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 a certain quantity of lemon juice.

The feature model for this product line is depicted in Figure 3.1a. Feature `Vending Machine` is the root of the feature model, representing a product. Each of the valid products has the functionality of serving a beverage, represented by the mandatory feature `Beverage`. Its two child features, `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 lemon to a beverage is an optional behavior, it is modeled by the optional feature `Lemon`. If a product is generated with the feature selection {`Soda`} (i.e., `Lemon` is not selected), a possible model of its probabilistic behavior is depicted in Figure 3.1b. If the feature selection is {`Tea, Lemon`}, the derived product has a probabilistic behavioral model as in Figure 3.1c.




(a) Feature model

(b) Behavior for {`Soda`}

(c) Behavior for {`Tea, Lemon`}

Figure 3.1: Vending machine product line example

In both DTMC examples, 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 hand-offs. 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 $t_l$ correspond to the behavior of adding lemon to tea, that is, they only exist in products derived by a feature selection with both features `Tea` and `Lemon`.

We now discuss the modeling approaches we devised to leverage PMCs as representations of product-line reliability. These approaches build on the assumptions we made for DTMC models of user-oriented software reliability, which are established in Section 2.2:

- Reliability of a software system can be interpreted in a binary fashion: either the system outputs a correct result (*success*) or it fails (*error*).
  - If the system is intended to perform an infinite loop (e.g., the Body Sensor Network [76]), one may consider the correctness of a single iteration.
- States in the model represent the execution of a function by some system module.
  - Thus, the state space is finite.
- Module failures do not propagate to one another.
- DTMCs are time-homogeneous—i.e., the transition probabilities do not change over time.
- Parallelism and nondeterminism are not considered.

The way by which we represent variability as PMCs and generate products (i.e., DTMCs) from the resulting variable assets is classified according to the current accepted taxonomy [3, 47] in two main categories: annotation-based (or annotative) and composition-based (or compositional). Each of these kinds of models will play a role in the analysis strategies presented in Section 3.2. We also present a correspondence between compositional and annotative models in Section 3.2.4.

### 3.1.1 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 3.2, 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.[1] 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 lemon to tea, using the variable $t_l$. Some states of the model for serving tea, as well as the behaviors corresponding to `Soda` and its lemon-adding variant, are omitted for brevity. The whole model can be seen in Figure B.1.

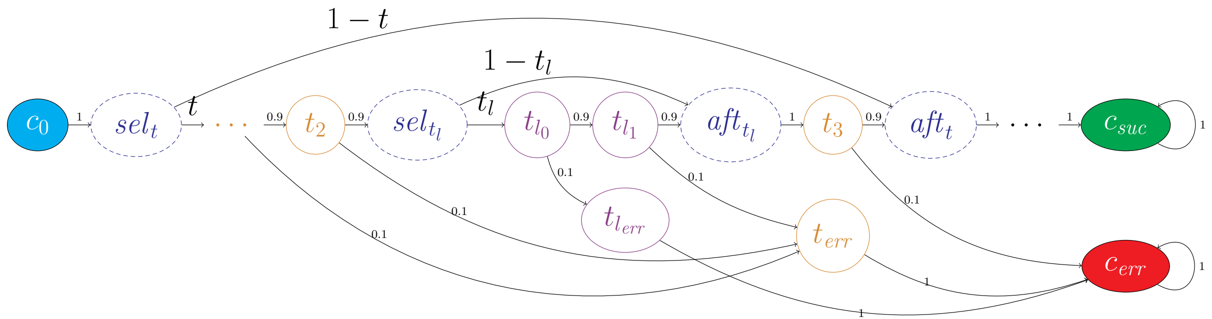



Figure 3.2: Annotative PMC for the vending machine

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

**Definition 5** (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. $\forall_{s' \in S} \cdot \mathbf{P}(s, s') \in [0, 1] \wedge \mathbf{P}(s, S) = 1$ (the probabilities of all outgoing transitions are constants that add up to 1); or
2. $\exists_{s_0, aft_s \in S}\, \exists_{x \in X} \cdot Succ(s) = \{s_0, aft_s\} \wedge \mathbf{P}(s, s_0) = x \wedge \mathbf{P}(s, aft_s) = 1 - x$ (there are exactly two outgoing transitions, whose probabilities are expressed as a single variable and its complement).

[1] The states $sel_t$ and $aft_t$ are analogous to the `#ifdef` and `#endif` macros of the C preprocessor, usually seen in preprocessor-based product lines.

The states in Figure 3.2 that fall in the second case are $sel_t$ and $sel_{t_l}$ (as well as $sel_s$ and $sel_{s_l}$, 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 (i.e., ones whose transitions have invalid probabilities or that violate 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 behavioral models. Thus, let $p_x$ be the presence condition for the behavior identified by $x$. In our vending machine example, we would have $p_t = \texttt{Tea}$, $p_{t_l} = \texttt{Tea} \wedge \texttt{Lemon}$, $p_s = \texttt{Soda}$, and $p_{s_l} = \texttt{Soda} \wedge \texttt{Lemon}$. 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 6** (Presence function)**.** Given a set $X$ of variables and a feature model $FM$, a presence function is a function $p : X \to [\![FM]\!] \to \mathbb{B}$ such that, for all $x \in X$ and all $c \in [\![FM]\!]$,

$$p(x)(c) = \begin{cases} 1 & \text{if } c \models p_x \quad \textit{(presence condition is satisfied)} \\ 0 & \text{otherwise} \end{cases}$$

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

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 8** (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 5);
- $FM$ is a feature model;
- $p : X \rightarrow [\![FM]\!] \rightarrow \mathbb{B}$ is a presence function (Definition 6); and
- $w$ is an evaluation factory (Definition 7) 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 8) and of the presence function $p$ (Definition 6) 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 for uniformity, since it is the annotative counterpart of the composition factory $w'$ in a compositional probabilistic model (Definition 18). The definitions of the presence function and the annotative evaluation factory are only similar because the set of Real values in the image of the possible evaluations (i.e., $\{0, 1\}$) in the annotative case correspond to our Real encoding of Boolean values.

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 (which is the basis for product derivation) without resorting to the just defined concept of annotative models.

**Definition 9** (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 $\pi : PMC_X \times \big([\![FM]\!] \to X \to \mathbb{R}\big) \times [\![FM]\!] \to DTMC$ is such that

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

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

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 8, $\mathcal{P}$ is an annotative PMC (Definition 5), so states with no variability (case 1) satisfy the needed properties by definition. For states $s$ with variability (case 2), it holds that

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

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 8, 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 [39], metaproduct [84], variant simulator [91], or product simulator [2].

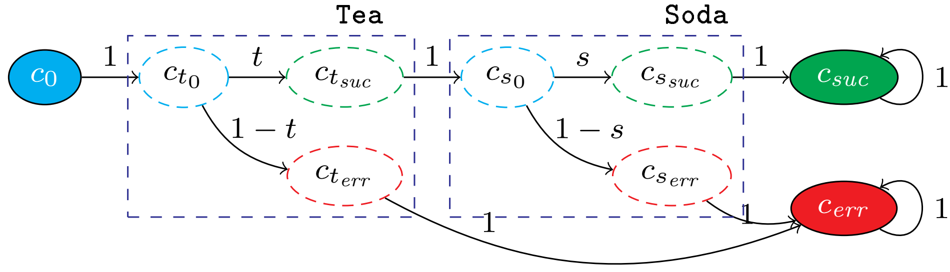
### 3.1.2 Compositional Models

A compositional representation of variable configuration-specific behavior consists of a hierarchy 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.

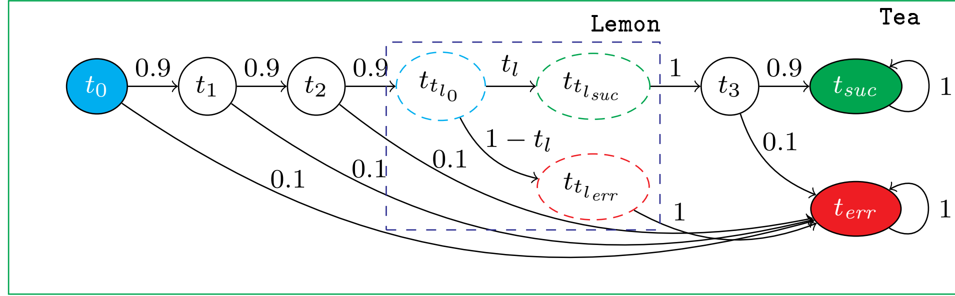
Figure 3.3 illustrates this concept. For the vending machine example, the *top-level* PMC $\mathcal{P}_\top$ would be as in Figure 3.3a. In this PMC, we introduce triples of dashed states that act as placeholders for the abstracted behavior. We call these states and corresponding transitions *slots*. For instance, the top-level PMC in Figure 3.3a 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 3.2), 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.

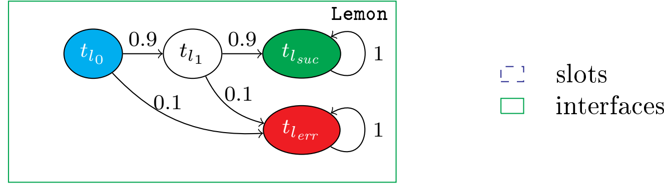
Figure 3.3b shows the PMC $\mathcal{P}_t$ for the tea behavior, in which we use a slot to abstract the optional lemon-adding behavior, whose behavior is modeled by the PMC $\mathcal{P}_{t_l}$ in Figure 3.3c. Since this tea-lemon PMC has no variability, it is in fact a regular DTMC. We




(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 $\mathcal{P}_{t_l}$ for the behavior of adding lemon to tea

Figure 3.3: Compositional PMCs for the vending machine

omit the PMCs for serving soda ($\mathcal{P}_s$) and for adding lemon to soda ($\mathcal{P}_{s_l}$), for brevity, but the complete example can be seen in Figure B.2 (Appendix B).

This example depicts compositional PMCs as being generated by extracting subgraphs from a DTMC model of system reliability. The intuition about *PMC composition* is that it represents the reverse process: "inlining" PMCs back into the slots from which they were extracted in the first place. Although composition in the context of DTMCs and transition systems may be interpreted as *parallel* composition (using operators such as interleaving and handshaking), this is not the case. We use this concept here in the sense of composition-based variability modeling.

Formally, we define a compositional PMC as a PMC in which transition probabilities depend on the value of some probabilistic reachability property of other PMCs. For a PMC defined this way, possible evaluations map variables to real numbers within the interval $[0, 1]$, instead of the binary set $\{0, 1\}$ used for an annotative model (see Definition 7). To compose PMCs modeled this way with one another, we augment the definition of a PMC

with explicit mentions of *success* and *error* states.

**Definition 10** (Compositional PMC)**.** A compositional PMC $\mathcal{P}$ is a tuple $(S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$, where:

- $S$ is a set of states, $s_0 \in S$ is the initial state, $X$ is a set of variables, and $\mathbf{P}$ is a transition probability matrix, such that $(S, s_0, X, \mathbf{P}, T)$ is an annotative PMC (see Definition 5).

- States $s_{suc}, s_{err} \in S$ are called *success* and *error* states, respectively. Together with the initial state, $s_0$, they define the *interface* of the compositional PMC: $interface(\mathcal{P}) = \{s_0, s_{suc}, s_{err}\}$ (solid box around PMCs in Figure 3.3).

- $T = \{s_{suc}\}$. That is, $s_{suc}$ is the only target state.

- The success and error states are the only *bottom strongly connected components* [7] in $\mathcal{P}$, that is:

  - once one of them is reached, no other state is ever reachable; and

  - they are the only states satisfying this property.

  This restriction ensures that we model all executions as either successful (if the success state is reached) or non-successful (if the error state is reached).

Definition 10 builds on Definition 5 to define the *structure* of compositional PMCs, but the intended semantics of variables in this type of parametric Markov chain is different from the corresponding semantics in an annotative PMC. In a compositional PMC, the condition that the outgoing transitions of a given node are either all constant or all variable (inherited from Definition 5) relates to the concept of *slots*, whereas annotative PMCs treat variable transitions as behavioral switches. Informally, a slot for the variable $x$ (dashed boxes in Figure 3.3) marks the part of a product's behavior where a configuration-specific behavior (identified by $x$) takes place. Note that there can be more than one slot for a given behavior.

**Definition 11** (Compositional PMC slot)**.** For a compositional PMC $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$, a *slot* for $x \in X$ is a triple $(s_{x_0}, s_{x_{suc}}, s_{x_{err}})$, where:

- $s_{x_0}, s_{x_{suc}}, s_{x_{err}} \in S$;

- $Succ(s_{x_0}) = \{s_{x_{suc}}, s_{x_{err}}\}$;

- $\mathbf{P}(s_{x_0}, s_{x_{suc}}) = x \wedge \mathbf{P}(s_{x_0}, s_{x_{err}}) = 1 - x$.

The set of slots for $x$ in $\mathcal{P}$ is denoted by $slots^{\mathcal{P}}(x)$, and the set of states belonging to any slot in $slots^{\mathcal{P}}(x)$ is given by $slotStates^{\mathcal{P}}(x) = \{s \in S \mid \exists_{\varsigma \in slots^{\mathcal{P}}(x)} \cdot s \in \varsigma\}$. We extend these definitions for the set of all slots in $\mathcal{P}$ for any variable in $X$ ($slots^{\mathcal{P}}(X)$) and the set of states belonging to any slot in that set ($slotStates^{\mathcal{P}}(X)$).

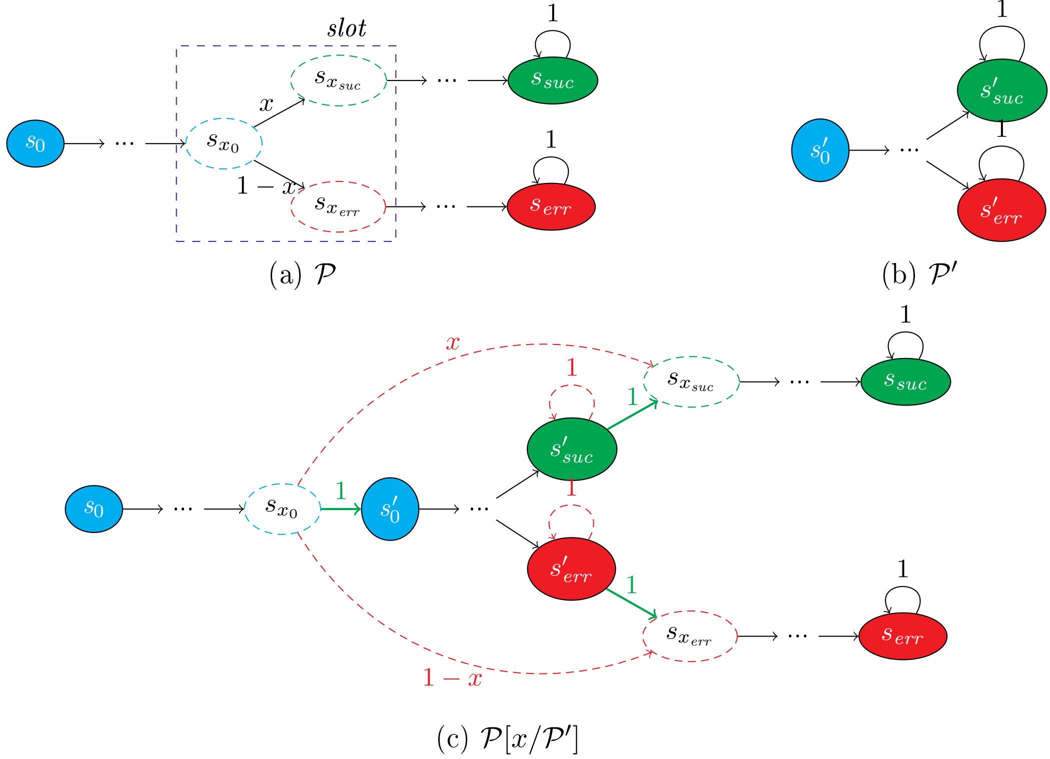
With compositional PMCs at hand, we need to be able to derive a DTMC, modeling the behavior of a given product of the product line, as in Section 3.1.1. Before we can handle the product-line aspect, we must define the mechanics of PMC composition. The intuition is that composition is achieved by connecting the interface (solid outer box) of a compositional PMC $\mathcal{P}'$ to the slots (dashed boxes) in a compositional PMC $\mathcal{P}$ that are meant to abstract the behavior in $\mathcal{P}'$, that is, $slots^{\mathcal{P}}(x)$ (see Figure 3.4).

**Definition 12** (Partial PMC composition)**.** Given a compositional PMC $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$ and a variable $x \in X$, assume that $x$ occurs only once in $\mathcal{P}$, and let $\mathcal{P}' = (S', s'_0, s'_{suc}, s'_{err}, X', \mathbf{P}', T')$ be a compositional PMC to be composed on that single slot marked by $x$. The partial PMC composition $\mathcal{P}[x/P']$ is a compositional PMC $\mathcal{P}'' = (S'', s''_0, s''_{suc}, s''_{err}, X'', \mathbf{P}'', T'')$ such that:

- $S'' = S \uplus S'$, where $\uplus$ denotes the disjoint union operator (all states are disjointly merged);
- $s''_0 = s_0$, $s''_{suc} = s_{suc}$, and $s''_{err} = s_{err}$ (the interface of $\mathcal{P}$ is preserved);
- $X'' = X \setminus \{x\} \cup X'$ (the occurrence of $x$ is replaced by a copy of $\mathcal{P}'$, whose variables are those of $X'$);
- $T'' = T$ (target states of the base PMC are preserved);
- $\mathbf{P}''$ is such that
  - $\mathbf{P}''(s_{x_0}, s'_0) = 1$ (new transition from a slot's initial state to the initial state of the corresponding composed PMC)
  - $\mathbf{P}''(s'_{suc}, s_{x_{suc}}) = 1$ (new transition from the success state of a composed PMC to the success state of the corresponding slot)
  - $\mathbf{P}''(s'_{err}, s_{x_{err}}) = 1$ (new transition from the error state of a composed PMC to the error state of the corresponding slot)
  - $\mathbf{P}''(s_{x_0}, s_{x_{suc}}) = 0$ (slot's success transition is removed)
  - $\mathbf{P}''(s_{x_0}, s_{x_{err}}) = 0$ (slot's error transition is removed)
  - $\mathbf{P}''(s'_{suc}, s'_{suc}) = 0$ (success loops from composed PMCs are removed)
  - $\mathbf{P}''(s'_{err}, s'_{err}) = 0$ (error loops from composed PMCs are removed)

– For all remaining combinations of $s_1, s_2 \in S''$:

$$\mathbf{P}''(s_1, s_2) = \begin{cases} \mathbf{P}(s_1, s_2) & \text{if } s_1, s_2 \in S \setminus \mathit{slotStates}^{\mathcal{P}}(x) \\ \mathbf{P}'(s_1, s_2) & \text{if } s_1, s_2 \in S' \\ 0 & \text{otherwise} \end{cases}$$




(a) $\mathcal{P}$ (b) $\mathcal{P}'$

(c) $\mathcal{P}[x/\mathcal{P}']$

Figure 3.4: Example of a partial composition of two PMCs

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 3.4c, which depicts the partial composition of the compositional PMC $\mathcal{P}'$ (Figure 3.4b) into $\mathcal{P}$ (Figure 3.4a) from the perspective of a single slot. New transitions are green bold, while red dashed transitions are the ones suppressed during composition. We say this transformation is partial because slots for variables other than $x$ are not subject to composition.

Since there might be more than one slot for a given variable, we extend the concept of partial composition to mean the composition of $n$ renamings of a given compositional PMC $\mathcal{P}'$ into each of the $n$ slots for a single variable $x$ in another compositional PMC $\mathcal{P}$. A full (*total*) composition is then obtained by composing PMCs over all slots in a given base compositional PMC at once. Such a composition relies on a *composition function*—

a function $u' : X \to \mathscr{P}$ that yields a compositional PMC $\mathcal{P} \in \mathscr{P}$ to compose in the corresponding slots for any given variable. The detailed definitions of PMC renaming (Definition 33) and total PMC composition (Definition 34) are presented in Appendix A.3.

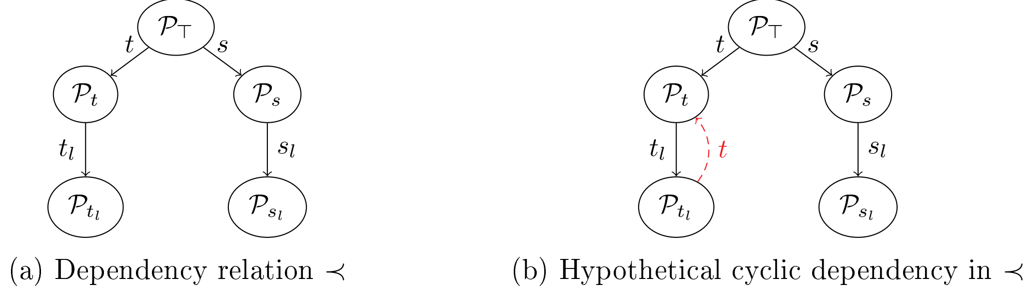
In a composition, slots mark locations where behavioral model fragments (i.e., other compositional PMCs) can be inserted to expand the base behavior. However, nothing so far prevents composition to happen at arbitrary slots (e.g., composing the behavior of adding lemon to soda in the slot for $t$, which was meant to represent the behavior of serving tea). Thus, we need a way to relate slots and the intended abstracted configuration-specific behaviors. We do so by naming compositional PMCs with the same variables that are used in the slots that mark their places, by means of an *identifying function*.

**Definition 13** (Identifying function)**.** Let $\mathscr{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a finite set of compositional PMCs $\mathcal{P}_i$, each with a set $X_i$ of variables, where $i \in \{1, \ldots, n\}$. An identifying function is a bijection $idt : \mathscr{P} \to I$, where $I \supset \bigcup_{\mathcal{P}_i} X_i$ is a set of variables that contains all variables in the compositional PMCs $\mathcal{P}_i$.

Since $idt$ is a bijection, the set $I$ of identifiers must have the same cardinality as $\mathscr{P}$. In practical terms, we arbitrarily identify PMCs that do not directly correspond to an abstracted behavior (i.e., those that are not directly referred by variables in other PMCs). This is the case of top-level PMCs, which are mainly composed of states that are shared between the behaviors of all products. For the vending machine product line (Figure B.2, summarized in Figure 3.3), for which $\mathscr{P} = \{\mathcal{P}_\top, \mathcal{P}_t, \mathcal{P}_{t_l}, \mathcal{P}_s, \mathcal{P}_{s_l}\}$, we can define $I = \{\top, t, t_l, s, s_l\}$ and, correspondingly, $idt = \{\mathcal{P}_\top \mapsto \top, \mathcal{P}_t \mapsto t, \mathcal{P}_{t_l} \mapsto t_l, \mathcal{P}_s \mapsto s, \mathcal{P}_{s_l} \mapsto s_l\}$.

An identifying function induces a *dependency* relation over PMCs, based on their names and the variables they employ to abstract behavior in slots. If we denote this relation by $\prec$, in the vending machine example, we can say that $\mathcal{P}_{t_l} \prec \mathcal{P}_t \prec \mathcal{P}_\top$, meaning $\mathcal{P}_\top$ depends on $\mathcal{P}_t$, which, in turn, depends on $\mathcal{P}_{t_l}$. Also, $\mathcal{P}_{s_l} \prec \mathcal{P}_s \prec \mathcal{P}_\top$. Figure 3.5a illustrates this dependency relation as a dependency graph, in which edges are labeled according to the variables identifying the respective dependencies. There should be no infinite descending chain under this relation, because otherwise one would infinitely compose PMCs and never get a DTMC as a result. This could happen as a modeling error, for instance, as introduced by the hypothetical dashed red cyclic dependency in Figure 3.5b. Hence, we require the dependency relation among compositional PMCs to be *well-founded*, meaning there can be no infinite sequence $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \ldots$ such that $\forall_{i \geq 1} \cdot \mathcal{P}_{i+1} \prec \mathcal{P}_i$. This also prohibits cyclic dependencies, since they would allow infinite chains.

**Definition 14** (Dependency relation induced in compositional PMCs)**.** Given a finite set $\mathscr{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ of compositional PMCs $\mathcal{P}_i$, each with a set $X_i$ of variables, and



(a) Dependency relation $\prec$ (b) Hypothetical cyclic dependency in $\prec$

Figure 3.5: Dependency relation induced in the vending machine

a corresponding identifying function $idt : \mathscr{P} \to I$, the binary relation $\prec: \mathscr{P} \times \mathscr{P}$ is the well-founded dependency relation induced by $idt$ and by the use of variables in the $\mathcal{P}_i$. That is,

$$\forall_{\mathcal{P}_i,\mathcal{P}_j \in \mathscr{P}} \cdot idt(\mathcal{P}_j) \in X_i \Leftrightarrow \mathcal{P}_j \prec \mathcal{P}_i$$

We read $\mathcal{P}_j \prec \mathcal{P}_i$ as "$\mathcal{P}_i$ depends on $\mathcal{P}_j$".

A consequence of this definition is that, in a finite set of compositional PMCs with an identifying function, there must be, at least, one PMC that depends on no other and has no variability whatsoever (a *minimal* PMC), and, at least, one PMC on which no other depends (a *maximal* PMC). In the vending machine (Figure B.2), the minimal PMCs are $\mathcal{P}_{t_l}$ and $\mathcal{P}_{s_l}$, while $\mathcal{P}_\top$ is the single maximal PMC.

**Definition 15** (Minimal and maximal compositional PMCs)**.** Given a set $\mathscr{P}$ of compositional PMCs, an identifying function $idt$, and the corresponding induced well-founded relation $\prec$, a compositional PMC $\mathcal{P} \in \mathscr{P}$ is called *minimal* iff

$$\nexists_{\mathcal{P}' \in \mathscr{P}} \cdot \mathcal{P}' \prec \mathcal{P}$$

Conversely, $\mathcal{P} \in \mathscr{P}$ is called *maximal* iff

$$\nexists_{\mathcal{P}' \in \mathscr{P}} \cdot \mathcal{P} \prec \mathcal{P}'$$

Maximal PMCs can be seen as models of top-level behavior in a system, such as the main tasks usually represented by UML activity diagrams. In an automation software charged with managing different workflows, for instance, one could model each of the workflows as a separate behavior with internal variability, thus yielding as many maximal PMCs as there are tasks to accomplish. The number of maximal PMCs in a compositional model is mainly a modeling decision, and analyzing the whole product line amounts to

analyzing each of these top-level behaviors. Thus, without loss of generality, we consider models that have only one maximal PMC[2], which we denote by $\mathcal{P}_\top$.

After composition, the variability in a compositional PMC is replaced by the variabilities of the PMCs composed into it. That is to say, the set of variables of the resulting compositional PMC is given by $\bigcup_{i=1}^{k} X_i$, the set of variables in all composed PMCs. In the vending machine (Figure 3.3), for instance, if we compose the tea PMC $\mathcal{P}_t$ (Figure 3.3b) into the top-level PMC $\mathcal{P}_\top$ (Figure 3.3a) using the slot $(c_{t_0}, c_{t_{suc}}, c_{t_{err}})$, the resulting compositional PMC $\mathcal{P}_\top[t/\mathcal{P}_t]$ will no longer have variable $t$, but will have a new variable $t_l$, stemming from $\mathcal{P}_t$. Consequently, to derive a product, one has to recursively perform the composition operation until a plain DTMC is returned.

This recursive approach to derive a product by composition relies on an identifying function *idt* to assign PMCs to slots corresponding to their identifiers. This composition depends upon satisfaction of a presence condition. Thus, before we can properly define this approach of *derivation by composition*, we must define how to proceed with composition in the case that the presence condition of a model to be composed is not satisfied. We achieve this result by composing the *feature disabler* compositional PMC, depicted in Figure 3.6. This compositional PMC models an always successful behavior, so composing it would not affect the overall reliability of the base model.

**Definition 16** (Feature disabler compositional PMC)**.** The feature disabler compositional PMC, $\mathcal{P}_\perp = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$, is a compositional PMC such that:

- $S = \{s_0, s_{suc}, s_{err}\}$
- $X = \emptyset$
- $\mathbf{P}(s_0, s_{suc}) = 1$, $\mathbf{P}(s_{suc}, s_{suc}) = 1$, and $\mathbf{P}(s_{err}, s_{err}) = 1$. Otherwise, for $s, s' \in S$, $\mathbf{P}(s, s') = 0$.
- $T = \{s_{suc}\}$

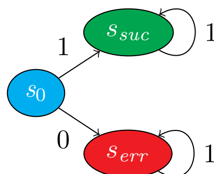



Figure 3.6: Feature disabler compositional PMC $\mathcal{P}_\perp$

[2]The existence of minimal and maximal PMCs follows from the well-foundedness of $\prec$. More details are available at Appendix A.1.

Similar to what we have achieved with evaluation factories (Definition 7), we need to constrain the possible compositions to ones that respect both: (a) satisfying presence conditions and (b) matching of slots and compositional PMCs via an identifying function. To enable this, we define a *composition factory* as a higher-order function that constrains compositions based on possible configurations of the modeled product line. This is the basis of product derivation.

**Definition 17** (Composition factory)**.** Given a set $\mathscr{P}$ of compositional PMCs, a set $I$ of identifiers that is a superset of the variables used in slots, and a feature model $[\![FM]\!]$, a composition factory $w' : [\![FM]\!] \to I \to DTMC$ is a function that, for a given configuration $c \in [\![FM]\!]$, yields a *composition function* $w'(c) : I \to DTMC$.

To populate this definition with concrete composition factories, we fix the set $I$ of identifiers as well as an identifying function, thus inducing a dependency relation that establishes which models should be composed to get a probabilistic model for a desired product. This way, a *compositional model* of a product line is a set of compositional PMCs closed under this dependency relation.

**Definition 18** (Compositional probabilistic model)**.** A compositional probabilistic model for a product line with feature model $FM$ is a tuple $(\mathscr{P}, \prec, I, idt, p, w', FM)$, where:

- $\mathscr{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ is a finite set of compositional PMCs $\mathcal{P}_i = (S_i, s_{i_0}, s_{i_{suc}}, s_{i_{err}}, X_i, \mathbf{P}_i, T_i)$ (Definition 10).
- $I$ is a set of variables, such that $I \supset \bigcup_{\mathcal{P}_i} X_i$ and $|I| = |\mathscr{P}|$. These variables are a superset of all variables in the compositional PMCs in $\mathscr{P}$.
- $idt : \mathscr{P} \to I$ is an identifying function for $\mathscr{P}$ (Definition 13).
- $\prec : \mathscr{P} \times \mathscr{P}$ is the well-founded dependency relation induced by $idt$ and by the use of variables in the compositional PMCs $\mathcal{P}_i$ (Definition 14).
- $FM$ is a feature model.
- $p : I \to [\![FM]\!] \to \mathbb{B}$ is a presence function (Definition 6) denoting presence conditions satisfaction.
- $w'$ is a composition factory (Definition 17) recursively defined as

$$w'(c)(x) = \begin{cases} \mathcal{P}_i[x_1/w'(c)(x_1), \ldots, x_k/w'(c)(x_k)] & \text{if } p(x)(c) = 1 \\ \mathcal{P}_\perp & \text{otherwise} \end{cases}$$

  where $\mathcal{P}_i \in \mathscr{P}$, $idt(\mathcal{P}_i) = x \in I$, and $X_i = \{x_1, \ldots, x_k\}$.

This definition allows us to model the behavior of a product line in a compositional way. To leverage this model for product-line analysis, we define a way to derive a DTMC that is consistent with the behavior of a product generated using the same configuration.

**Definition 19** (Derivation by composition)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and a compositional PMC $\mathcal{P} \in \mathscr{P}$ with a set $X$ of variables, the DTMC derivation by composition $\pi'(\mathcal{P}, w', c)$ is defined as

$$\pi'(\mathcal{P}, w', c) = \mathcal{P}[X/w'(c)]$$

The notation is overloaded from PMC evaluation, since both are model transformations that operate on variables. Since $w'$ is defined recursively, we need to guarantee its execution terminates, which is why we require $\prec$ to be well-founded. The termination proof (Lemma 11) is presented in Appendix A.2.

## 3.2 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 [85]. Following the taxonomy of Thüm et al. [85], we discussed possible strategies for each of the variability representations (annotative and compositional) presented in Section 3.1.

Figure 3.7 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, Figure 3.7 is a commuting diagram (as we will demonstrate later in this section), meaning that different analysis paths are equivalent (i.e., they yield equal results) 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 $\pi$) and then analyzed (function $\alpha$), or variability-aware analysis is applied, using some form of parametric model checking (function $\hat{\alpha}$). The first choice yields a product-based strategy (Section 3.2.1), 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 $\sigma$), yielding feature-product-based (Section 3.2.3) and family-product-based (Section 3.2.2) strategies; or to interpret the expressions in terms of ADDs (function *lift*), effectively evaluating them for the whole family of models at once (function $\hat{\sigma}$)—a step we call *expression lifting*. The latter represents feature-family-based (Section 3.2.3) and family-based (Section 3.2.2) strategies.

As an example of walking through the choices of Figure 3.7, 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.

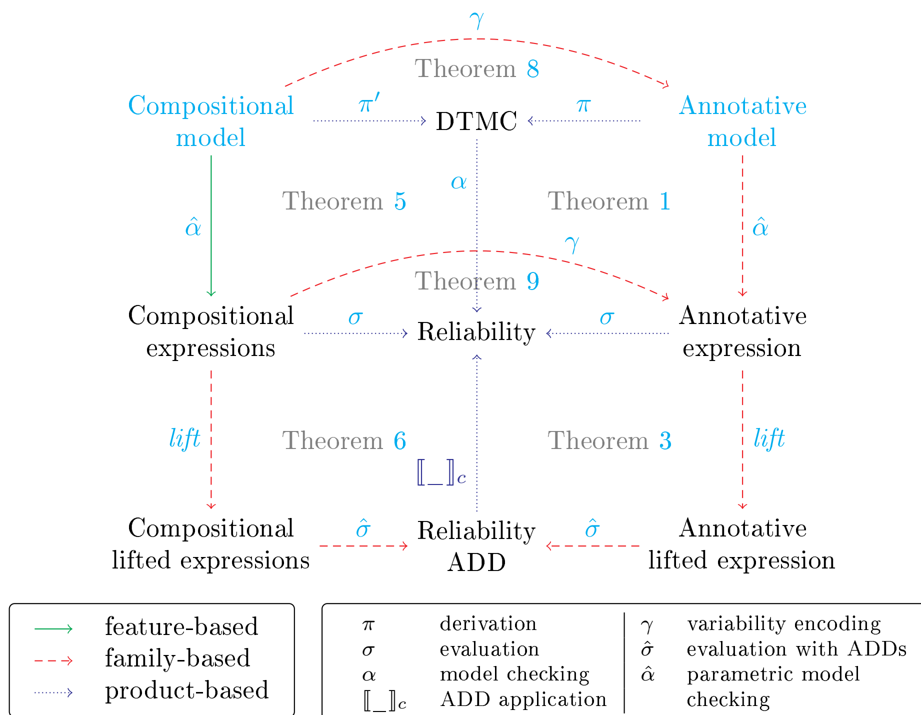



Figure 3.7: Commutative diagram of product-line reliability analysis strategies

Note that the commuting diagram in Figure 3.7 presents links to the definitions of models and analysis steps. Each section in this diagram also provide a pointer to the theorem stating its commutativity. For instance, Theorem 5 states that the upper left quadrant of the diagram commutes. The remaining theorems are omitted from that figure to reduce visual clutter.

In the remaining sections, we detail each of our strategies and analysis steps with the goal of making statements about their commuting relations. Section 3.2.1 presents product-based analysis strategies for both annotative and compositional models, with the goal of establishing a baseline for the remaining soundness proofs. Section 3.2.2 discusses family-product-based and family-based analyses of annotative models. Feature-product-based and feature-family-based analysis strategies are the subject of Section 3.2.3, which focuses on compositional models. Then, Section 3.2.4 bridges the gap between analyses of annotative and compositional models (function $\gamma$ in Figure 3.7), establishing their commutativity. Finally, we leverage these results to present the novel feature-family-product-based strategy in Section 3.2.5.

### 3.2.1 Product-based Strategies

Product-based analysis strategies are based on the analysis of generated products or models thereof [85]. In Section 3.1, we have discussed how to represent probabilistic behavioral models of product lines as PMCs, using both annotative and compositional approaches. There, we also described how to derive models of individual products, both for the annotative and the compositional approaches. 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 $\alpha$.

**Definition 20** (Non-parametric model checking)**.** The non-parametric model checking step $\alpha : DTMC \rightarrow [0,1]$ consists of applying the algorithm by Hahn et al. [41]. 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, $\alpha$ yields constant functions, which we interpret as plain Real numbers.

Although there are other algorithms for reliability model checking of regular (non-parametric) DTMCs, we use the algorithm by Hahn et al. [41] 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 and compositional models.

**Strategy 1** (Product-based analysis of annotative models)**.** Given an annotative model $(\mathcal{P}, p, w, FM)$, a product-based analysis yields, for all $c \in [\![FM]\!]$,

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

or, alternatively,

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

**Strategy 2** (Product-based analysis of compositional models)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$, a product-based analysis yields, for all $c \in [\![FM]\!]$,

$$\alpha(\pi'(\mathcal{P}_\top, w', c))$$

where $\mathcal{P}_\top$ is the maximal PMC in $\mathscr{P}$ under $\prec$.

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

Both analysis strategies presented in this section derive 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 strategies establish a baseline for proving the soundness of other strategies.

### 3.2.2 Family-based Strategies

According to Thüm et al. [85], 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 this is discussed later in Section 3.2.4.

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 3.2.2). Then, the aforementioned analysis is leveraged to build a pure family-based (i.e., non-enumerative) strategy (Section 3.2.2). 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.

### Family-product-based Strategy

A family-product-based strategy is a family-based strategy followed by a product-based strategy over intermediate results [85]. 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 $\hat{\alpha}$, where the $\hat{}$ symbol denotes that it is a variability-aware version of the non-parametric model checking function $\alpha$ (Definition 20).

**Definition 21** (Parametric model checking)**.** The parametric model checking analysis step $\hat{\alpha} : PMC_X \to \mathcal{F}_X$ consists of applying the algorithm by Hahn et al. [41] 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 $\sigma$.

**Definition 22** (Expression evaluation)**.** Given an expression $\varepsilon$ 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 $\sigma$ 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.

**Strategy 3** (Family-product-based analysis)**.** Given an annotative model $(\mathcal{P}, p, w, FM)$, the family-product-based analysis yields, for all $c \in [\![FM]\!]$,

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

or, alternatively,

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

Figure 3.8 illustrates the family-product-based strategy in contrast with the product-based one (Section 3.2.1), providing an intuition for why they commute. DTMC derivation $\pi$ and expression evaluation $\sigma$ 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 $x$ was absent (i.e., $c \not\models p_x$), then the reliability would be 0.99.

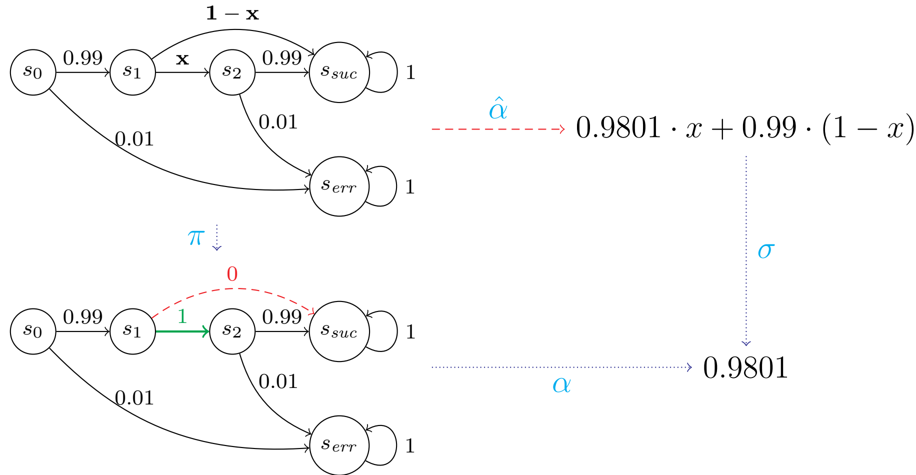



Figure 3.8: Example of family-product-based analysis ($\hat{\alpha}$ followed by $\sigma$) in contrast to a product-based analysis ($\pi$ followed by $\alpha$) of an annotative PMC, for a configuration satisfying $x$'s presence condition

To be considered sound, a family-product-based analysis must be equivalent[3] 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.

**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*

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

[3]Whenever two analysis strategies yield equal reliability values, we say they are *r-equivalent*.

*Proof.*

$$\begin{aligned} \alpha(\mathcal{P}[X/u]) &= \alpha(\mathcal{P}_u) && \text{(syntax change)} \\ &= Pr^{\mathcal{P}_u}(s_0, T) && \text{(Definition 20)} \end{aligned}$$

and, since $u$ is well-defined,

$$= \hat{\alpha}(\mathcal{P})[X/u] \qquad \text{(Lemma 1 and Definition 21)}$$

□

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* $c \in [\![FM]\!]$

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

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

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

$$\begin{aligned} \alpha([\![\mathcal{P}]\!]_c^w) &= \alpha(\mathcal{P}[X/w(c)]) && \text{(Definition 9)} \\ &= \hat{\alpha}(\mathcal{P})[X/w(c)] && \text{(Lemmas 2 and 3)} \\ &= [\![\hat{\alpha}(\mathcal{P})]\!]_c^w && \text{(Definition 22)} \end{aligned}$$

□

As a major result, Theorem 1 states that the diagram in Figure 3.9 commutes. This diagram corresponds to the upper right quadrant in Figure 3.7.

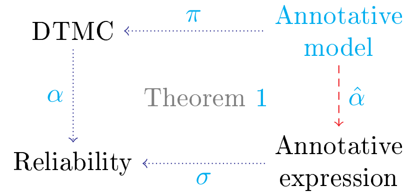



Figure 3.9: Statement of Theorem 1

#### 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 reliability 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, which 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* [93]. 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 B.1) as an example. Its reliability expression after parametric model checking has 8 terms, one of which is $0.124659 \cdot t \cdot t_l$. Starting from the evaluation factory $w$, we can derive functions $\psi_x$ that, for each variable $x$, take a configuration $c \in [\![FM]\!]$ as input and output the corresponding value $w(c)(x)$. For $t$ and $t_l$, for instance, these functions would be as follows:

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

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 $\hat{r}(c) = 0.124659 \cdot \psi_t(c) \cdot \psi_{t_l}(c)$ (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 23** (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 $\varepsilon$, 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 $\varepsilon$ is a rational expression over Real constants and variables $x_i \in X$, $|X| = n$, $A_1, \ldots, A_n$ are ADDs, and $\hat{\varepsilon} = lift(\varepsilon)$, 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 induction on the structure of the rational expression $\varepsilon$. The base cases are constant expressions and single variables, for which the lemma holds. We then use induction and algebraic manipulation to prove for the arithmetic case (i.e., $\varepsilon = \varepsilon_1 \odot \varepsilon_2$, where $\odot \in \{+, -, \times, \div\}$) and for exponentiation. Proof details can be found in Appendix A.4. □

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 24** (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 25** (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, $\hat{\sigma}$, is defined as

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

*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 $\varepsilon$ 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 $c \in [\![FM]\!]$,*

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

*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 $c \in [\![FM]\!]$,

$$\begin{aligned}
\hat{\sigma}(\hat{\varepsilon}, \hat{w})(c) &= \hat{\varepsilon}[X/\hat{w}](c) && \text{(Definition 25)} \\
&= \hat{\varepsilon}[x_1/\hat{w}(x_1), \ldots, x_n/\hat{w}(x_n)](c) \\
&= \varepsilon[x_1/\hat{w}(x_1)(c), \ldots, x_n/\hat{w}(x_n)(c)] && \text{(Lemma 4)} \\
&= \varepsilon[x_1/w(c)(x_1), \ldots, x_n/w(c)(x_n)] && \text{(Definition 24)} \\
&= \varepsilon[X/w(c)] \\
&= \sigma(\varepsilon, w, c) && \text{(Definition 22)}
\end{aligned}$$

□

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.

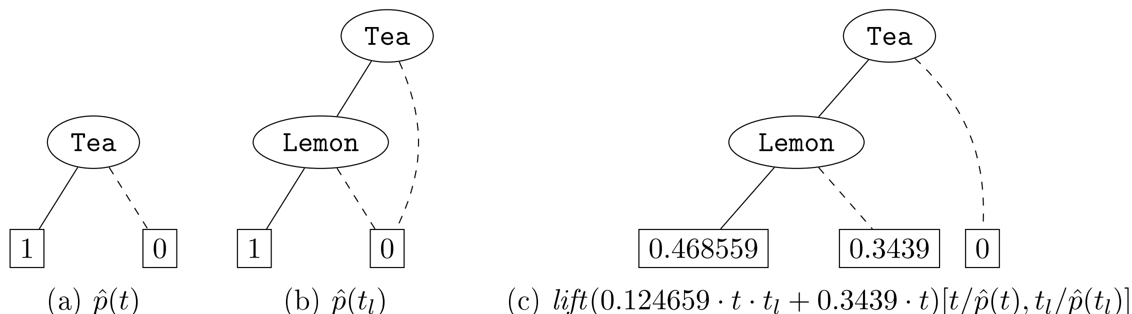
**Lemma 5** (Soundness of lifted annotative evaluation factory). *Given an annotative model $(\mathcal{P}, p, w, FM)$ and a function $\hat{p} : X \to (\mathbb{B}^{|FM|} \to \mathbb{B})$ that encodes presence conditions for variables as ADDs, then $\hat{w} = \hat{p}$ is a lifted evaluation factory for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

*Proof.* From Definition 8, 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 $t_l$ are, respectively, `Tea` and `Tea` $\wedge$ `Lemon`. Then, the ADDs $\hat{p}(t)$ and $\hat{p}(t_l)$ are given by the Figures 3.10a and 3.10b, where we use the notation presented in Section 2.3. If we evaluate a lifted version of the example expression $\varepsilon = 0.124659 \cdot t \cdot t_l + 0.3439 \cdot t$ (2 terms from the actual reliability expression for the vending machine annotative model in Figure B.1) with $\hat{p}$, the resulting ADD will be $\hat{r} = 0.124659 \cdot \hat{p}(t) \cdot \hat{p}(t_l) + 0.3439 \cdot \hat{p}(t)$, as depicted in Figure 3.10c. Hence, for a given configuration $c \in [\![FM]\!]$, if both `Tea` and `Lemon` are present (i.e., $\hat{p}(t)(c) = 1$ and $\hat{p}(t_l)(c) = 1$), 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 `Lemon` are absent, then $\hat{r}(c) = 0$.




(a) $\hat{p}(t)$ (b) $\hat{p}(t_l)$ (c) $\mathit{lift}(0.124659 \cdot t \cdot t_l + 0.3439 \cdot t)[t/\hat{p}(t), t_l/\hat{p}(t_l)]$

Figure 3.10: 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* $(\mathcal{P}, p, w, FM)$, $\varepsilon = \hat{\alpha}(\mathcal{P})$, *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* $c \in [\![FM]\!]$

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

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

*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, $[\![\hat{\sigma}(\hat{\varepsilon}, \hat{p})]\!]_c = [\![\varepsilon]\!]_c^w$. □

Figure 3.11 illustrates the main result from Theorem 3. The depicted diagram, which corresponds to the lower right quadrant in Figure 3.7, is commutative because of this theorem.

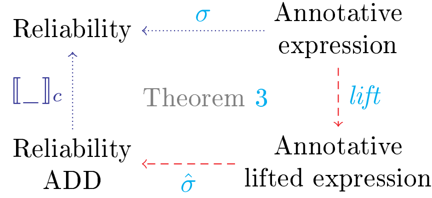



Figure 3.11: Statement of Theorem 3

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

**Strategy 4** (Family-based analysis)**.** Given an annotative model $(\mathcal{P}, p, w, FM)$, a family-based analysis yields

$$\hat{\sigma}\big(\mathit{lift}(\hat{\alpha}(\mathcal{P})), \hat{p}\big)$$

The result of a family-based analysis is a Boolean function encoded as an ADD. Such an 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*

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

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

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

□

As a key result, Theorem 4 states that the diagrams in Figure 3.12 commute. Both diagrams correspond to the right half of the one in Figure 3.7.

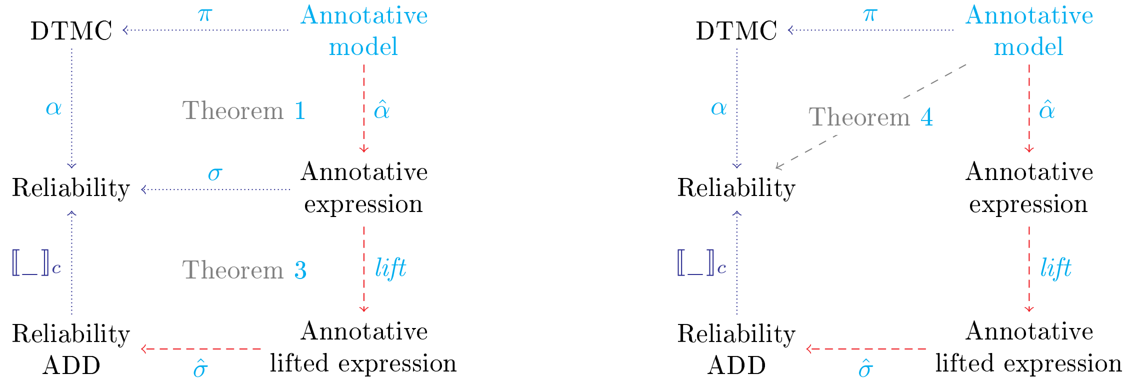



Figure 3.12: Alternative views of the statement of Theorem 4

### 3.2.3 Feature-based Strategies

A feature-based analysis strategy is one that (a) operates only on domain artifacts and that (b) analyzes the artifacts belonging to each feature in isolation [85]. Compositional models describe modular behaviors that represent units of variability. A given PMC within a compositional model may represent the behavior associated with one or more features, or even model part of a given feature's behavior (in case of behavior scattering). In this sense, analyzing individual PMCs of a compositional model can be seen as analyzing features in isolation, which is why we use this kind of probabilistic model to discuss feature-based strategies. Moreover, since our focus is on reliability, which is highly influenced by feature interactions, we cannot use a pure feature-based strategy [85]. Thus, we concentrate on feature-product-based and feature-family-based analysis strategies.

Similar to what happens with family-based strategies (Section 3.2.2), the feature-family-based approach builds upon concepts used by the feature-product-based strategy, and performing one or the other is a matter of choosing product-based or family-based

analysis steps after a preliminary feature-based step. Because of that, we first discuss the feature-product-based strategy (Section 3.2.3), focusing on the feature-based step of applying parametric model checking to each compositional PMC to generate corresponding compositional expressions. These reliability expressions can be evaluated for every possible configuration, yielding a product-based step and giving rise to a feature-product-based strategy. Alternatively, we can lift each expression and evaluate them using ADDs, in a similar fashion to what we did for the family-based strategy (Section 3.2.2). This leads to an overall feature-family-based strategy, which we discuss in Section 3.2.3.

### Feature-product-based Strategy

A product-line analysis strategy is feature-product-based (a) if it consists of a feature-based analysis followed by a product-based analysis and (b) if the analysis results of the feature-based analysis are used in the product-based analysis [85]. The preliminary feature-based analysis step consists of applying the parametric model checking function $\hat{\alpha}$ to each PMC in a compositional model, yielding corresponding reliability expressions. These resulting expressions preserve the dependency relation, since each of them is defined in terms of the same variables as its originating PMC and can be assigned the same identifier.

As an example, the compositional model of the vending machine product line (Figure 3.3) yields the following expressions after the feature-based analysis step: $\hat{\alpha}(\mathcal{P}_\top) = 1 \cdot t \cdot s$, $\hat{\alpha}(\mathcal{P}_t) = 0.6561 \cdot t_l$, and $\hat{\alpha}(\mathcal{P}_{t_l}) = 0.81$. Also, $\hat{\alpha}(\mathcal{P}_s) = 0.729 \cdot s_l$ and $\hat{\alpha}(\mathcal{P}_{s_l}) = 0.81$ for the remaining PMCs in Figure B.2.

A bottom-up evaluation of variables can be applied for each valid configuration, giving rise to the product-based analysis step. This procedure consists of *compositional expression evaluation*, that is, expression evaluation using a *compositional evaluation factory* derived from the composition factory used for the corresponding PMCs.

**Definition 26** (Compositional evaluation factory)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$, a compositional evaluation factory is defined as an evaluation factory (Definition 7) $w : [\![FM]\!] \to I \to \mathbb{R}$, such that for all $c \in [\![FM]\!]$ and $x \in I$,

$$w(c)(x) = \begin{cases} \sigma(\hat{\alpha}(\mathcal{P}), w, c) & \text{if } p(x)(c) = 1 \\ 1 & \text{otherwise} \end{cases}$$

where $idt(\mathcal{P}) = x$. Alternatively, we can write

$$w(c)(x) = \begin{cases} [\![\hat{\alpha}(\mathcal{P})]\!]_c^w & \text{if } p(x)(c) = 1 \\ 1 & \text{otherwise} \end{cases}$$

In other words, whereas a composition factory composes a recursively derived version of PMC $\mathcal{P}'$ into slots identified by a variable $x$ of a PMC $\mathcal{P}$, a compositional evaluation factory composes a recursively evaluated version of $\hat{\alpha}(\mathcal{P}')$ in every occurrence of the variable $x$ in $\hat{\alpha}(\mathcal{P})$. This recursion always terminates, because $\prec$ is a well-founded relation (see Lemma 12, in Appendix A.2).

We define the feature-product-based analysis of compositional models as a recursive evaluation of the expressions obtained from the feature-based step, using the compositional evaluation factory shown above. This recursion starts from the maximal PMC in the compositional model, traversing the dependency graph induced by $\prec$ (Figure 3.5a) in a depth-first fashion.

For the vending machine product line (Figure 3.3), for instance, the computation for configuration $c = \{\texttt{Tea}, \texttt{Lemon}\}$ would be as follows: Starting with $\hat{\alpha}(\mathcal{P}_\top)$, we evaluate the presence conditions for its variables, $t$ and $s$. Since $p_s = \texttt{Soda}$ is not satisfied, $s$ is evaluated to 1, ending the computation for this branch. On the other hand, $p_t = \texttt{Tea}$ is satisfied, so we step into this branch to compute $\hat{\alpha}(\mathcal{P}_t)$ under $c$. The only variable in this expression, $t_l$, has its presence condition satisfied by $c$, so we step further into this branch to compute $\hat{\alpha}(\mathcal{P}_{t_l})$ under $c$. Since this expression denotes a constant value, we return this value and the recursion terminates, yielding the following constant expression:

$$[\![\hat{\alpha}(\mathcal{P}_\top)]\!]_c = 1 \cdot \underbrace{\left(0.6561 \cdot \overbrace{(0.81)}^{[\![\hat{\alpha}(\mathcal{P}_{t_l})]\!]_c}\right)}_{[\![\hat{\alpha}(\mathcal{P}_t)]\!]_c} \cdot \underbrace{(1)}_{[\![\hat{\alpha}(\mathcal{P}_s)]\!]_c}$$

We generalize and formalize this procedure as follows.

**Strategy 5** (Feature-product-based analysis)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and the compositional evaluation factory $w$, derived from the composition factory $w'$, the feature-product-based analysis yields, for all $c \in [\![FM]\!]$,

$$\sigma(\hat{\alpha}(\mathcal{P}_\top), w, c)$$

or, alternatively,

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

where $\mathcal{P}_\top$ is the maximal PMC in $\mathscr{P}$ under the dependency relation $\prec$.

To establish the soundness of the feature-product-based strategy, we need to compare it to the product-based strategy for compositional models. We state this result in the following theorem.

**Theorem 5** (Soundness of feature-product-based analysis)**.** *Given a compositional model* $(\mathscr{P}, \prec, I, idt, p, w', FM)$*, for all configurations* $c \in [\![FM]\!]$*, it holds that*

$$\sigma(\hat{\alpha}(\mathcal{P}), w, c) = \alpha(\pi'(\mathcal{P}, w', c))$$

*or, alternatively,*

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

*where* $\mathcal{P} \in \mathscr{P}$ *and* $w$ *is the compositional evaluation factory (Definition 26) derived from the composition factory* $w'$*.*

*Proof.* We use well-founded induction. The base of the induction is when $\mathcal{P}$ is minimal with respect to $\prec$. Since minimal PMCs have empty sets of variables, $\pi'(\mathcal{P}, w', c) = \mathcal{P}$ and $\hat{\alpha}(\mathcal{P}) = \alpha(\mathcal{P})$. Thus, the statement holds for the base case.

The general case is proved by expanding definitions in the proof goal and applying the induction hypothesis and Lemma 3. The complete proof is presented in Appendix A.3. □

> As a further major result, Theorem 5 states that the diagram in Figure 3.13 commutes. This diagram relates to the upper left quadrant in Figure 3.7.

Compositional model —$\pi'$→ DTMC
$\hat{\alpha}$ ↓ Theorem 5 ↓ $\alpha$
Compositional expressions —$\sigma$→ Reliability

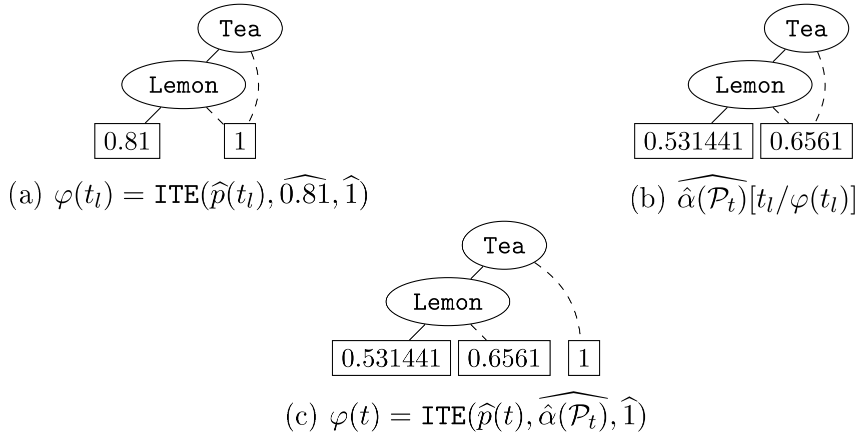
Figure 3.13: Statement of Theorem 5

### Feature-family-based Strategy

Similar to the family-based strategy (Section 3.2.2), the feature-family-based strategy leverages ADDs to store and reason about variational data. Since the preceding feature-based analysis yields expressions over reliabilities, this variational data is made of Real values corresponding to the reliabilities of the products of a product line. Again, lifting

expressions involves lifting the corresponding evaluation factory. In this process, the presence conditions are encoded in ADDs to represent the variability under feature selection. This encoding is achieved by the ADD operator ITE (*if-then-else*).

Let us revisit expression evaluation in the vending machine example (Figure 3.3). We have seen the expression for $t_l$ is the constant 0.81, so its lifted version is the constant ADD $\widehat{0.81}$ (according to the notation introduced in Definition 23). The expression for $t$, $\hat{\alpha}(\mathcal{P}_t) = 0.6561 \cdot t_l$, has the variable $t_l$. Thus, if the presence condition $p_{t_l} = \texttt{Tea} \land \texttt{Lemon}$ is satisfied, this variable must be evaluated to the constant value 0.81, assuming the value 1 otherwise. Thus, the lifted expression $\widehat{\hat{\alpha}(\mathcal{P}_t)}$ is evaluated with an ADD encoding this choice, given by $\varphi(t_l) = \texttt{ITE}(\widehat{p}(t_l), \widehat{0.81}, \widehat{1})$ and depicted in Figure 3.14a. The evaluated lifted expression $\widehat{\hat{\alpha}(\mathcal{P}_t)}[t_l/\varphi(t_l)]$ is the ADD product of the constant $\widehat{0.6561}$ and $\varphi(t_l)$, shown in Figure 3.14b. The procedure is repeated for every composition, so that the variable $t$ in the expression $\widehat{\hat{\alpha}(\mathcal{P}_\top)}$ would be replaced by the ADD in Figure 3.14c, which already encodes the combined presence conditions for $t$ and $t_l$.




(a) $\varphi(t_l) = \texttt{ITE}(\widehat{p}(t_l), \widehat{0.81}, \widehat{1})$

(b) $\widehat{\hat{\alpha}(\mathcal{P}_t)}[t_l/\varphi(t_l)]$

(c) $\varphi(t) = \texttt{ITE}(\widehat{p}(t), \widehat{\hat{\alpha}(\mathcal{P}_t)}, \widehat{1})$

Figure 3.14: Example of lifted compositional expression evaluation

The function $\varphi$ shown in the example is the lifted version of the compositional evaluation factory $w$. We first present a formal definition of $\varphi$ and then proceed to proving its soundness. Soundness of the feature-family-based strategy follows from this result and from the soundness of the feature-product-based strategy (Section 3.2.3).

**Definition 27** (Lifted compositional evaluation factory)**.** Given a compositional probabilistic model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and the compositional evaluation factory $w$, derived from the composition factory $w'$, the lifted evaluation factory $\varphi : I \to (\mathbb{B}^{|FM|} \to \mathbb{R})$ is a function that, for any $x \in I$, yields an ADD $\varphi(x)$ such that:

$$\varphi(x) = \texttt{ITE}(\hat{p}(x), \widehat{\hat{\alpha}(\mathcal{P})}[X/\varphi], \hat{\mathbf{1}})$$

where $\mathcal{P} \in \mathscr{P}$, $idt(\mathcal{P}) = x$, $\widehat{\hat{\alpha}(\mathcal{P})} = lift(\hat{\alpha}(\mathcal{P}))$ and $\hat{\mathbf{1}}$ is the constant ADD corresponding to the function $(c \in [\![FM]\!]) \mapsto 1$.

The next lemma, which is the compositional counterpart of Lemma 5, states this function $\varphi$ is indeed a lifted version of $w$.

**Lemma 6** (Soundness of lifted compositional evaluation factory). *Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and the compositional evaluation factory $w$, derived from the composition factory $w'$ (Definition 26), for all $x \in I$ and all $c \in [\![FM]\!]$ it holds that*

$$\varphi(x)(c) = w(c)(x)$$

*Proof.* We first expand the definitions of $\varphi$ (Definition 27) and $w$ (Definition 26), then proceed to compare corresponding cases. The cases in which the presence condition is not satisfied are trivially equal; for the complementary case, we use well-founded induction on the dependency relation $\prec$, along with the soundness result for expression lifting (Lemma 4). The reader is invited to follow the complete proof in Appendix A.4. □

This way, the ADDs yielded by function $\varphi$ from Definition 27 correctly encode the variation in values returned by the compositional evaluation factory $w$. An immediate consequence is that the expressions resulting from the feature-based analysis step can, indeed, be lifted and then evaluated using $\varphi$, and this gives us the same results as the corresponding (i.e., for the same configurations) product-based evaluations. This is expressed by the following theorem.

**Theorem 6** (Soundness of expression evaluation using $\varphi$). *Given a compositional probabilistic model $(\mathscr{P}, \prec, I, idt, p, w', FM)$, the compositional evaluation factory $w$, derived from the composition factory $w'$, and $x \in I$, let $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$ be such that $idt(\mathcal{P}) = x$, $\mathcal{P} \in \mathscr{P}$. If $\varepsilon = \hat{\alpha}(\mathcal{P})$, $\hat{\varepsilon} = lift(\varepsilon)$, and $\varphi$ is the lifted compositional evaluation factory obtained from $w$ (Definition 27), then, for all $c \in [\![FM]\!]$, it holds that*

$$\hat{\varepsilon}[X/\varphi](c) = \varepsilon[X/w(c)]$$

*Proof.* For the given compositional probabilistic model, Lemma 6 states $\varphi$ is a sound lifted counterpart of $w$. Hence, by Theorem 2, $\varepsilon[X/w(c)] = \hat{\varepsilon}[X/\varphi](c)$. In other words, $[\![\hat{\sigma}(\hat{\varepsilon}, \varphi)]\!]_c = [\![\varepsilon]\!]_c^w$. □

So, Theorem 6 states that the diagram in Figure 3.15 commutes. This diagram corresponds to the lower left quadrant in Figure 3.7.

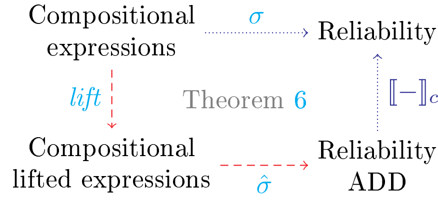



Figure 3.15: Statement of Theorem 6

The feature-family-based analysis strategy leverages the preceding results to yield an ADD encoding all reliabilities for valid configurations of the product line. This process is formally defined as follows.

**Strategy 6** (Feature-family-based analysis)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and the lifted compositional evaluation factory $\varphi$, derived from $w'$, the feature-family-based strategy yields

$$\hat{\sigma}\big(lift(\hat{\alpha}(\mathcal{P}_\top)), \varphi\big)$$

where $\mathcal{P}_\top$ is the maximal PMC in $\mathscr{P}$ under the dependency relation $\prec$.

Similar to the family-based strategy, the feature-family-based strategy is sound if this ADD is such that applying it to every valid configuration $c \in [\![FM]\!]$ results in the same probability as if the original compositional model had been derived for $c$ and the resulting DTMC had been model-checked for probabilistic reachability (product-based strategy). The difference is that, in the feature-family-based case, this statement holds for every PMC in the compositional model.

**Theorem 7** (Soundness of feature-family-based analysis)**.** *Given a compositional model* $(\mathscr{P}, \prec, I, idt, p, w', FM)$ *and the lifted compositional evaluation factory* $\varphi$*, derived from* $w'$*, for every PMC* $\mathcal{P} \in \mathscr{P}$ *and for all configurations* $c \in [\![FM]\!]$ *it holds that*

$$[\![\hat{\sigma}\big(lift(\hat{\alpha}(\mathcal{P}_\top)), \varphi\big)]\!]_c = \alpha([\![\mathcal{P}]\!]_c^{w'})$$

*Proof.* Let $w$ be the compositional evaluation factory derived from the composition factory $w'$. The proof follows from successive application of Theorems 5 and 6:

$$\begin{aligned} [\![\hat{\sigma}\big(lift(\hat{\alpha}(\mathcal{P})), \varphi\big)]\!]_c &= [\![\hat{\alpha}(\mathcal{P})]\!]_c^{w} && \text{(Theorem 6)} \\ &= \alpha([\![\mathcal{P}]\!]_c^{w'}) && \text{(Theorem 5)} \end{aligned}$$

□

As a key result, Theorem 7 states that the diagrams in Figure 3.16 commute. Both diagrams correspond to the left half of the one in Figure 3.7.

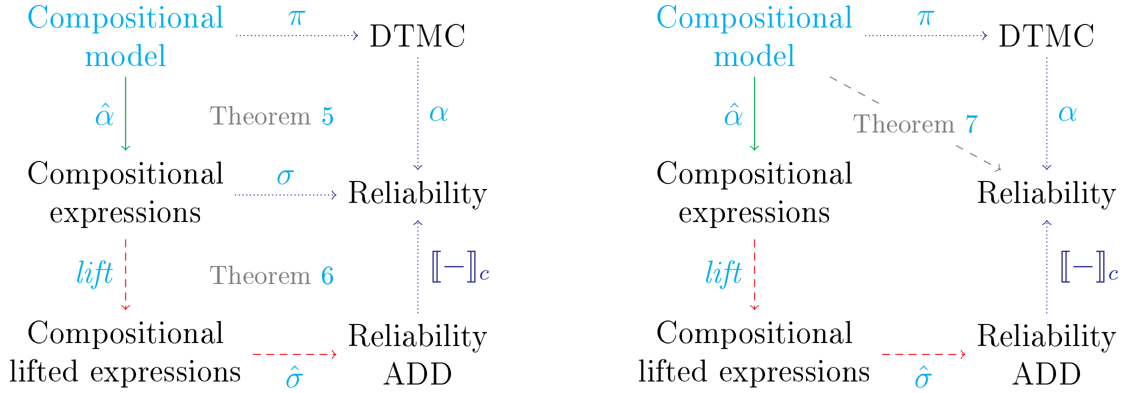



Figure 3.16: Alternative views of the statement of Theorem 7

### 3.2.4 Bridging Compositional and Annotative Models

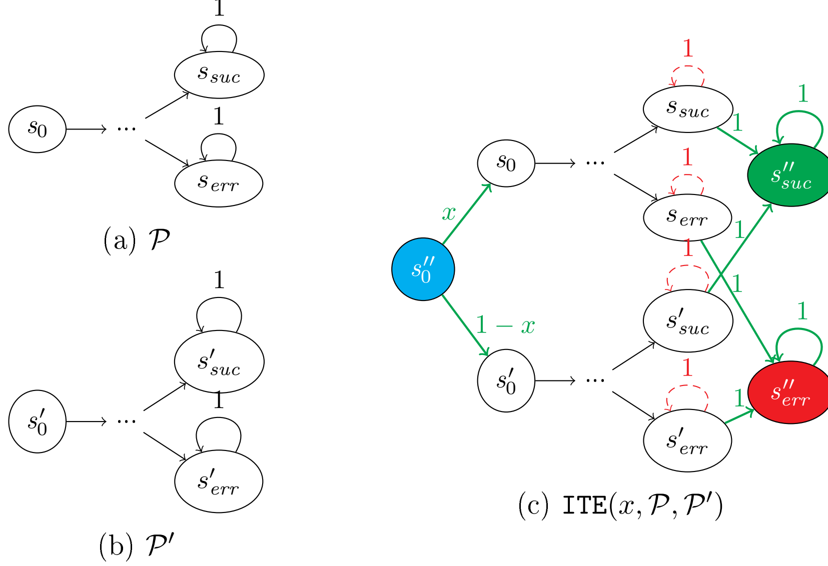
Thus far, we have discussed family-based analysis strategies applied to annotative models and feature-based analysis strategies applied to compositional models. We now present a technique to transform any composition-based model into an r-equivalent annotation-based model. This ability may be useful in the case that the reliability analysis of a given product line is predictably more efficient if performed using a strategy suited for annotative models, such as our family-product-based and family-based approaches. This transformation of models resembles *variability encoding* techniques, that is, the rewriting of compile-time variability as load-time or run-time variability [2, 3, 74, 91].

Although the concepts of compilation and execution are not defined for Markov chains, variability encoding, as established in the literature, has the main goal of creating artifacts that can be analyzed by off-the-shelf tools. Correspondingly, we are able to transform a compositional model, which cannot be directly model-checked (because it is split into a number of PMCs), into an annotative model, which can be immediately issued to a parametric model checker. Thus, we address the transformation of compositional models into annotative ones in terms of two *variability encoding functions*: one operating on PMCs (Section 3.2.4) and the other for handling expressions (Section 3.2.4).

#### Variability Encoding of PMCs

In terms of Markov chains, variability encoding can be realized by turning compositional models into annotative ones. This means transforming both the underlying compositional

PMCs and the composition factory $w'$ into a single annotative PMC with a corresponding evaluation factory. To accomplish this, we propose an *if-then-else* operator for PMCs that switches between possible states with a Boolean variable.




(a) $\mathcal{P}$

(b) $\mathcal{P}'$

(c) $\texttt{ITE}(x, \mathcal{P}, \mathcal{P}')$

Figure 3.17: Example ITE operator for PMCs

For brevity, the formal definition of this operator (Definition 35) is available in Appendix A.5.1. We rely on Figure 3.17 for intuition. Again, green bold arrows represent new transitions, whereas red dashed ones are removed. Intuitively, an evaluation that maps $x$ to 1 yields a PMC with the same behavior as $\mathcal{P}$ (consequent), while an evaluation that maps $x$ to 0 yields a PMC with the same behavior as $\mathcal{P}'$ (alternative). We formalize this behavioral switching in terms of r-equivalence.

**Lemma 7** (r-equivalence for ITE)**.** *Given two compositional PMCs, $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$ and $\mathcal{P}' = (S', s'_0, s'_{suc}, s'_{err}, X', \mathbf{P}', T')$, and a variable $x \notin X \cup X'$, let $\mathcal{P}'' = \texttt{ITE}(x, \mathcal{P}, \mathcal{P}')$. If $(\mathcal{P}'', p, w, FM)$ is an annotative model with $\mathcal{P}''$ as its underlying PMC*[4]*, where $p$, $w$, and $FM$ are arbitrarily chosen, then, for every $c \in [\![FM]\!]$,*

$$\alpha([\![\texttt{ITE}(x, \mathcal{P}, \mathcal{P}')]\!]^w_c) = \begin{cases} \alpha([\![\mathcal{P}]\!]^w_c) & \textit{if } p(x)(c) = 1 \\ \alpha([\![\mathcal{P}']\!]^w_c) & \textit{otherwise} \end{cases}$$

*Proof.* We are interested in computing the probability of reaching $s''_{suc}$ from $s''_0$ in $\mathcal{P}'' = \texttt{ITE}(x, \mathcal{P}, \mathcal{P}')$ under evaluation $w(c)$. Using the formal definition of ITE (Definition 35) and Property 1, we are able to derive a reachability expression with only two terms, each

[4]By Definition 10, any compositional PMC is also an annotative PMC (Definition 5). Thus, a compositional PMC can be the underlying PMC of an annotative model.

corresponding to the "activated" PMC ($\mathcal{P}$ or $\mathcal{P}'$). The complete proof can be found in Appendix A.5.1. □

The previous lemma establishes that the ITE operator has the effect of alternating behaviors if the resulting PMC is evaluated by replacing the switching variable $x$ with 0 or 1. With this result, we define the variability encoding of PMCs as a composition of PMCs using the ITE operator in a recursive way, with minimal PMCs as the base case. The alternative choice (second argument to ITE) is always the feature disabler PMC $\mathcal{P}_\perp$ (Definition 16), meaning no probabilistic behavior is actually added if the presence condition is not satisfied. This is coherent with the corresponding case in a composition factory (see Definition 18).

**Definition 28** (Variability encoding function for PMCs)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and $\mathcal{P}, \mathcal{P}_1, \ldots, \mathcal{P}_k \in \mathscr{P}$ such that $\mathcal{P}_i \prec \mathcal{P}$ and $x_i = idt(\mathcal{P}_i)$ for $i \in \{1, \ldots, k\}$, the variability encoding function $\gamma$ is defined as the following derivation by composition (Definition 19):

$$\gamma(\mathcal{P}) = \mathcal{P}[x_1/\texttt{ITE}(x_1, \gamma(\mathcal{P}_1), \mathcal{P}_\perp), \ldots, x_k/\texttt{ITE}(x_k, \gamma(\mathcal{P}_k), \mathcal{P}_\perp)]$$

This recursion terminates, since the arguments to the recursive calls involved are less than the input with respect to the well-founded relation $\prec$ (Lemma 11). Nonetheless, each variable $x_i$, which was meant as a slot marker, is replaced by a variable with the same name, but different meaning (i.e., intended to be evaluated with presence values). Since all variables in the PMC yielded by $\gamma$ have this issue, the composition factory from the original compositional model will no longer be suitable. Thus, we must broaden the scope of variability encoding to also transform the composition factory $w'$ into an annotative evaluation factory.

**Definition 29** (Variability encoding of PMCs)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$, let $\mathcal{P} \in \mathscr{P}$ be a PMC. Then, $(\gamma(\mathcal{P}), p, w, FM)$ is an annotative model that encodes $\mathcal{P}$'s variability, where $w$ is an evaluation factory as in Definition 8.

The main goal of variability encoding is to transform a compositional model into an annotative one, but this technique can only be exploited if the reliability analyses of both the original and the transformed models yield the same results. This fact is established by the following theorem.

**Theorem 8** (r-equivalence of variability encoding and derivation by composition)**.** *Given a compositional model* $(\mathscr{P}, \prec, I, idt, p, w', FM)$ *and* $\mathcal{P} \in \mathscr{P}$*, let* $(\gamma(\mathcal{P}), p, w, FM)$ *be its*

*variability-encoded annotative model. Then, for all $c \in [\![FM]\!]$,*

$$\alpha([\![\gamma(\mathcal{P})]\!]_c^w) = \alpha(\pi'(\mathcal{P}, w', c))$$

*Proof.* We use well-founded induction. For minimal PMCs (base of induction), $\gamma(\mathcal{P}) = \mathcal{P}$, so $[\![\gamma(\mathcal{P})]\!]_c^w = \mathcal{P}$. Likewise, $\pi'(\mathcal{P}, w', c) = \mathcal{P}$, so the proposition holds trivially.

As induction hypothesis, we have that $\alpha([\![\gamma(\mathcal{P}_i)]\!]_c^w) = \alpha(\pi'(\mathcal{P}_i, w', c))$ for all $\mathcal{P}_i \in \mathscr{P}$ such that $\mathcal{P}_i \prec \mathcal{P}$. Expanding $\alpha([\![\gamma(\mathcal{P})]\!]_c^w)$ and using previous soundness and r-equivalence results, we leverage this induction hypothesis to reach $\alpha(\pi'(\mathcal{P}, w', c))$.

The detailed proof can be found in Appendix A.5.1. □

In summary, Theorem 8 establishes the commuting diagram in Figure 3.18, which corresponds to the upper arc in Figure 3.7. Note that the derived DTMCs are not necessarily equal in a syntactic and structural sense—this theorem only states that $\alpha$ computes the same reliability for both models.

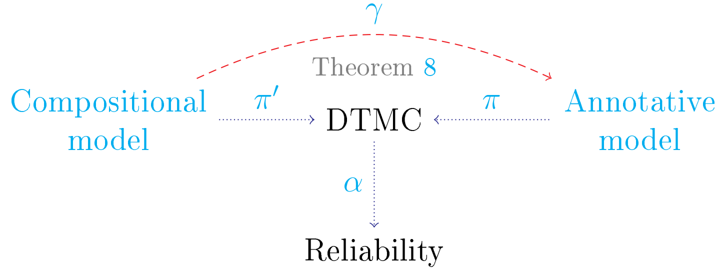



Figure 3.18: Statement of Theorem 8

The result from Theorem 8 indicates that, under the assumptions we made for user-oriented reliability models, the compositional variability representation is at least as expressive as the annotation-based one. Hence, either compositional or annotative models can be used to analyze a given product line. The decision on which one is more appropriate for each situation is not in the scope of this research; nonetheless, we conjecture that the modeling approach should follow the variability representation used in the system being modeled, as a means to mitigate the risk of introducing errors.

### Variability Encoding of Expressions

Aside from encoding variability in Markov chains, we can also encode variability in reliability expressions (represented by the arc in the middle row of Figure 3.7). Expressions derived from a compositional model can be combined to form a single larger expression (in

terms of operands). Applying such a transformation can be useful in cases where parsing and evaluating each compositional expression is less efficient than doing so for the single variability-encoded expression. As with PMCs, variability encoding of expressions can be defined in terms of a dedicated *if-then-else* operator for expressions.

**Definition 30** (ITE operator for expressions)**.** Given two expressions $\varepsilon$ and $\varepsilon'$ over the sets $X$ and $X'$ of variables, respectively, and a variable $x$, the *if-then-else* operator for expressions is defined as

$$\texttt{ITE}(x, \varepsilon, \varepsilon') = x \cdot \varepsilon + (1 - x) \cdot \varepsilon'$$

The set of variables of the resulting expression is $X'' = X \cup X' \cup \{x\}$. Additionally, $x$ is expected to be evaluated with a Boolean value, that is, 0 or 1. Procedures that do not affect the semantics of expressions, such as distributing the terms over the switching variable $x$ and simplifying the resulting expression, can be leveraged in working implementations.

This *if-then-else* operator merges two expressions to form a third one that uses a new variable to represent a choice and satisfies the following lemma.

**Lemma 8** (Extensional equality for expression ITE)**.** *Given two expressions $\varepsilon$ and $\varepsilon'$ over the sets $X$ and $X'$ of variables, respectively, and a variable $x$, let $X'' = X \cup X' \cup \{x\}$ and $u : X'' \to [0, 1]$ be an evaluation function such that $u(x) \in \mathbb{B}$. Then,*

$$\texttt{ITE}(x, \varepsilon, \varepsilon')[X''/u] = \begin{cases} \varepsilon[X/u] & \text{if } u(x) = 1 \\ \varepsilon'[X'/u] & \text{if } u(x) = 0 \end{cases}$$

*Proof.* We prove this by expanding the definition of ITE and performing algebraic manipulation. The complete proof can be found in Appendix A.5.2. □

The above lemma establishes that the ITE operator has the effect of alternating the semantics of the resulting expression between the ones of its arguments, but only if this resulting expression is evaluated with an evaluation that replaces the switching variable $x$ by 0 or 1. Similar to the ITE operator for PMCs, we define variability encoding of expressions as a composition of expressions using the ITE operator in a recursive way, with constant expressions (i.e., reliabilities of minimal PMCs) as the base case.

**Definition 31** (Variability encoding function for expressions)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and $\mathcal{P}, \mathcal{P}_1, \ldots, \mathcal{P}_k \in \mathscr{P}$ such that $\mathcal{P}_i \prec \mathcal{P}$ and $x_i = idt(\mathcal{P}_i)$ for $i \in \{1, \ldots, k\}$, let $\varepsilon = \hat{\alpha}(\mathcal{P})$ and $\varepsilon_i = \hat{\alpha}(\mathcal{P}_i)$. The variability encoding function $\gamma$ is overloaded for expressions as

$$\gamma(\varepsilon) = \varepsilon[x_1/\texttt{ITE}(x_1, \gamma(\varepsilon_1), \mathbf{1}), \ldots, x_k/\texttt{ITE}(x_k, \gamma(\varepsilon_k), \mathbf{1})]$$

This recursion terminates, since the arguments to the recursive calls involved are less than the input with respect to the well-founded relation $\prec$ (see Lemma 11).

Similar to variability encoding of PMCs, the new variables after encoding have the same names as the previous ones, but different meaning. Thus, we also transform the compositional evaluation factory $w$ (Definition 26) into an annotative evaluation factory (see Definition 8). This way, we ensure variables, which have all been transformed into conditionals, are evaluated as expected of the `ITE` semantics.

**Definition 32** (Variability encoding of expressions)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$, and the compositional evaluation factory $w$, derived from the composition factory $w'$, let $w_p$ be an annotative evaluation factory ($w$ in Definition 8) with the same presence conditions as $w$. That is, for all $c \in [\![FM]\!]$,

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

Then, for any $\mathcal{P} \in \mathscr{P}$ and $\varepsilon = \hat{\alpha}(\mathcal{P})$, $\gamma(\varepsilon)$ encodes $\varepsilon$'s variability under the evaluation $w_p$.

We state the soundness of variability encoding for expressions in terms of r-equivalence. For any configuration $c \in [\![FM]\!]$, a variability-encoded expression and its corresponding evaluation factory must yield the same reliabilities as the original compositional expressions and the corresponding compositional evaluation factory.

**Theorem 9** (Soundness of variability encoding for expressions)**.** *Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and $\mathcal{P}, \mathcal{P}_1, \ldots, \mathcal{P}_k \in \mathscr{P}$ such that $\mathcal{P}_i \prec \mathcal{P}$ and $x_i = idt(\mathcal{P}_i)$ for $i \in \{1, \ldots, k\}$, let $\varepsilon = \hat{\alpha}(\mathcal{P})$. Let also $w$ be the compositional evaluation factory derived from $w'$ (Definition 26) and $w_p$ be the annotative evaluation factory obtained from $w$ (Definition 32). Then, for all $c \in [\![FM]\!]$ it holds that*

$$\sigma(\gamma(\varepsilon), w_p, c) = \sigma(\varepsilon, w, c)$$

*Proof.* We use well-founded induction. For a minimal PMC $\mathcal{P}$ (base of induction), $\hat{\alpha}(\mathcal{P}) = \varepsilon$ has no variables. This way, $\gamma(\varepsilon) = \varepsilon$ and $\sigma(\varepsilon, u) = \varepsilon$ for any evaluation $u$. Thus, both sides of the equality evaluate to $\varepsilon$ and the proposition holds trivially.

As induction hypothesis, we have that $\sigma(\gamma(\varepsilon_i), w_p, c) = \sigma(\varepsilon_i, w, c)$ for all $\varepsilon_i = \hat{\alpha}(\mathcal{P}_i)$ such that $\mathcal{P}_i \prec \mathcal{P}$. Expanding $\sigma(\gamma(\varepsilon), w_p, c)$ and using previous soundness and extensional equality results, we leverage this induction hypothesis to reach $\sigma(\varepsilon, w, c)$.

The detailed proof can be found in Appendix A.5.2. □

As a further key result, Theorem 9 establishes the commuting diagram in Figure 3.19. This diagram corresponds to the arc in the middle section of Figure 3.7.

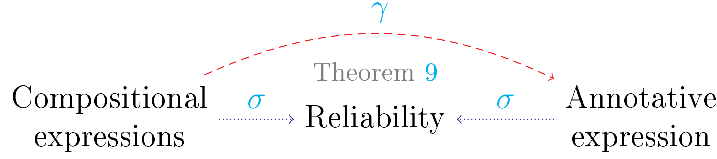



Figure 3.19: Statement of Theorem 9

**Variability Encoding of *Lifted* Expressions**

The symmetry exhibited by Figure 3.7 suggests that it makes sense to have arcs denoting variability encoding in all three levels of that diagram—i.e., not only for PMCs and expressions, but also for lifted expressions. However, lifting (Definition 23) operates on expression *semantics*, whereas variability encoding (Definition 32) operates on expression *syntax*. Because of that, variability encoding of lifted expressions is effectively the same operation as variability encoding of regular (Real-valued) rational expressions. Thus, we decided to represent this operation only once in our commuting diagram (Figure 3.7).

### 3.2.5 Feature-family-product-based Strategy

So far, we have proved that all compositions of analysis steps leading up to reliabilities in Figure 3.7 are r-equivalent. That is, these analysis steps commute, and, consequently, any path in this diagram can be equally taken to reach the same reliability value. By reflecting over the results condensed in this commuting diagram, we noticed a possible path that had not yet been exploited. This "unbeaten path", presented in Figure 3.20 as an excerpt from Figure 3.7, led us to derive a novel feature-family-product-based analysis strategy:

1. Starting from a compositional model (upper left corner), we apply parametric model checking ($\hat{\alpha}$) to obtain compositional expressions (*feature-based step*);

2. The resulting compositional expressions (lower left corner) are variability-encoded ($\gamma$) into a single annotative expression (*family-based step*); and

3. The annotative expression (lower right corner) is analyzed for each configuration $c \in [\![FM]\!]$ of the product line (*product-based step*).

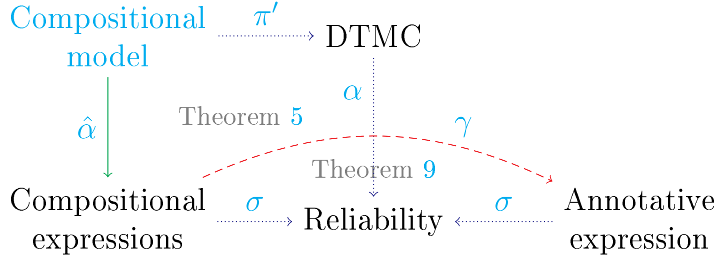



Figure 3.20: Commuting diagram leading to the feature-family-product-based strategy

The existence of a feature-family-product-based class of analyses was foreshadowed in a recent survey, but no instance has been found in the literature [85]. Thus, to the best of our knowledge, this is the first feature-family-product-based analysis to be presented, either formally or informally. The precise conditions under which this approach outperforms the others still need to be characterized by empirical studies. However, we believe it is an alternative to the family-product-based approach for cases in which (a) the model at hand is compositional and (b) applying variability encoding to the PMCs themselves is infeasible (e.g., the resulting annotative model is too big to be efficiently analyzed).

The novel strategy can be formally described as follows:

**Strategy 7** (Feature-family-product-based analysis)**.** Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and the compositional evaluation factory $w$, derived from the composition factory $w'$, the feature-family-product-based analysis yields, for all $c \in [\![FM]\!]$,

$$\sigma\big(\gamma(\hat{\alpha}(\mathcal{P}_\top)), w_p, c\big)$$

or, alternatively,

$$[\![(\gamma \circ \hat{\alpha})(\mathcal{P}_\top)]\!]_c^{w_p}$$

where $\mathcal{P}_\top$ is the maximal PMC in $\mathscr{P}$ under the dependency relation $\prec$, and $w_p$ is the variability-encoded annotative evaluation factory obtained from $w$ (Definition 32).

Since the diagram in Figure 3.7 commutes, this analysis is sound with respect to the product-based analysis of the same compositional model (Strategy 2). This soundness property is established by the following theorem:

**Theorem 10** (Soundness of feature-family-product-based analysis)**.** *Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and a compositional evaluation factory $w$, derived from the composition factory $w'$, for every PMC $\mathcal{P} \in \mathscr{P}$ and for all configurations $c \in [\![FM]\!]$ it holds that*

$$\sigma\big(\gamma(\hat{\alpha}(\mathcal{P})), w_p, c\big) = \alpha\big(\pi'(\mathcal{P}, w', c)\big)$$

*where $w_p$ is the variability-encoded annotative evaluation factory obtained from $w$ (Definition 32).*

*Proof.* The proof follows from successive application of other commutativity theorems.

$$\begin{aligned} \sigma\big(\gamma(\hat{\alpha}(\mathcal{P})), w_p, c\big) &= \sigma\big(\hat{\alpha}(\mathcal{P}), w, c\big) && \text{(Theorem 9)} \\ &= \alpha\big(\pi'(\mathcal{P}, w', c)\big) && \text{(Theorem 5)} \end{aligned}$$

□

In summary, Theorem 10 states that the diagram in Figure 3.21 commutes. This diagram corresponds to the upper left quadrant and the middle arc in Figure 3.7.

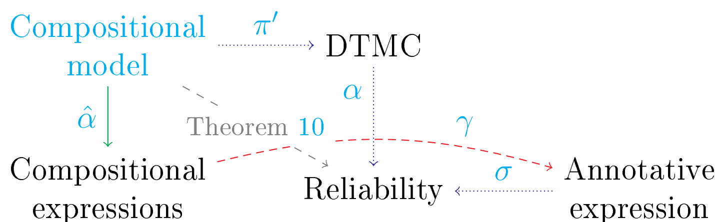



Figure 3.21: Statement of Theorem 10

## 3.3 Concluding Remarks

Together, the theorems demonstrated in this chapter constitute the main contribution of this work. Intermediate steps of the presented analysis techniques commute, making the diagram in Figure 3.7 fully commutative. Thus, any path constructed by following the arrows in that diagram yields an analysis that is r-equivalent to the one yielded by any other path that shares the same starting and ending points. This way, we guarantee all product-line 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. Such view, summarized by the commuting diagram in Figure 3.7, allows the organization and structuring of facts (e.g., commutativity of intermediate analysis steps) in a concise and precise manner, facilitating the communication of ideas. This contributes to a more comprehensive understanding of underlying principles used in these strategies, which we envision could help other researchers

to lift existing single-product analysis techniques to yet under-explored variability-aware approaches.

In what follows, we discuss the generality of our results and their application to related analysis strategies. We also discuss how this work relates to other topics of interest in product-line analysis—namely sampling strategies and detection of feature interactions.

**Generality:** Our results indicate that there is a general principle of using ADDs to encode variability in an efficient way. In our specific case, we used Real-valued ADDs to compute the values for the user-oriented reliability property of each possible product without resorting to a fully enumerative approach. Such efficiency comes from the algorithms for ADD binary operations [6], which are not restricted to Real numbers.

Hence, we believe that this technique may be explored to analyze other properties of product lines—especially numeric ones, such as performance and time-to-failure. In doing so, the bottom half of our commuting diagram (Figure 3.7) may be useful as a starting point, since it is dependent on rational expressions and evaluation factories, but not on the specific models (and properties) considered here.

Besides the general nature of ADD operations, we also believe that the commuting diagram itself may be used as a guide to specify similar theories for related domains. For instance, we can identify in Figure 3.7 that the central node (`Reliability`) represents the *property* under analysis, whereas the node just above that one (`DTMC`) represents the *model* for which that property can be analyzed. The upper left and upper right nodes (`Compositional Model` and `Annotative Model`) denote alternative ways to represent *variability* in such models, with inward arrows representing the binding of variability to *derive* a single product (in this case, a model thereof) and downward arrows denoting *variability-aware analysis*. Future work shall investigate how to leverage this framework to systematically generalize our results to other properties, models, and analysis techniques.

**Feature interactions:** When features of a product line are developed independently, it is possible that their combination causes unexpected behavior. To ensure the quality of products in a product line, such inadvertent feature interaction should be properly identified, managed, and resolved. Thus, the feature interaction problem is of particular interest when dealing with product-line analysis.

The combination of different dimensions of product-line analysis is, by itself, an approach to quality-checking in the presence of feature interactions [3]. In this particular work, we deal with models of user-oriented reliability. Although modeling techniques lie outside our scope, the analysis strategies formalized here assume that the reliability models use DTMC states to denote transfer of control between software modules (cf.

Section 2.2). Thus, whenever our analysis strategies combine models for valid product configurations, the results take into account the interaction of the *modeled* behavior for the different features.

The validity of this approach is dependent on the accurate modeling of software behavior. However, all resolutions of feature interactions can be abstracted as variants of a single pattern: implementing individual features and providing additional coordination logic to use them together [3]. Hence, we argue that the proper modeling of software behavior to reflect that implementation technique can lead to a sound feature-interaction-aware reliability analysis of a product line using our strategies.

**Sampling strategies:** The commutativity theory presented here establishes that our reliability analysis strategies yield the same results for any given *valid* configuration. This correspondence is enforced by the presence conditions encoded in evaluation and composition factories, as a means to activate/deactivate equivalent model fragments.

Thus, considering that sampling is a matter of checking a (suitable) subset of all products of a product line with a single-product analysis [3], we can say that the semantics of sampling strategies is contained in the semantics of the product-based strategy presented in Section 3.2.1. Therefore, sampling of *valid* configurations is covered by our formalization.

However, sampling strategies that allow arbitrary feature selections are also possible. Indeed, the benefits of such analysis strategies has been empirically assessed [59], especially as a means to detect feature interactions. Although our formalization as-is does not contemplate invalid configurations, doing so is a matter of extending the domains of presence functions (Definition 6) and evaluation factories (Definition 7) from $[\![FM]\!]$ to $2^F$ (where $F$ is a set of features and $FM$ is a feature model defined over $F$). Since $[\![FM]\!] \subseteq 2^F$ (by definition, cf. Section 2.1.1), it is possible to define such an extension that preserves the correspondence between model fragments for valid configurations and yet defines new correspondences for other feature combinations.

# Chapter 4

# Formalization in PVS

This chapter discusses the process of mechanizing the handcrafted version of our theory of commuting reliability analysis strategies [15]. We present an overview of the adopted specification strategy (Section 4.1) and a guide to the structure of the resulting specification (Section 4.2), whereby it is related to the original version. We also discuss the machine-assisted proof effort (Section 4.3) and reflect upon the mechanization process as a whole and the impact of design decisions (Section 4.4). Last, we discuss current limitations of the mechanized theory (Section 4.5).

Instead of a comprehensive description of every aspect of this mechanized theory, we only present in detail the representative constructs—either those that are extensively used throughout the specification or ones that embody design decisions for problems that are prone to more than one solution. Accordingly, we change some of the mechanized definitions (with respect to the actually implemented version) to improve readability.

The result so far is that all definitions, lemmas, and theorems in the original specification have been fully mechanized, but the machine-verified proofs of 5 of the new auxiliary lemmas are still in progress (Section 4.5). Moreover, some of the inner workings of discrete-time Markov chains are left unspecified by design, since the soundness of known facts about DTMCs is not debated and is also not the focus of this work.

The source code for the PVS specification is available at `https://github.com/thiagomael/rome-specs`. The *top* theory is in the PVS file rome, and serves both as a summary of the analysis strategies and as an entry point from which the whole specification can be checked (by issuing the command `pvs -batch -q -l batch-prove.el` from the directory containing the files). In this work, we use PVS version 6.0,[1] extended with the NASA PVS Library version 6.0.9.[2] It takes approximately 20 minutes for PVS to type-check and verify all proofs in the specification.[3] After this, the results will be

[1] `http://pvs.csl.sri.com/`

[2] `https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/`

available in the file `batch-rome.log`, including timing statistics and the status of all theories (at the end of that file).

## 4.1 Specification Strategy

To mechanize the manually specified theory, we followed the same order originally used when defining the different analysis strategies (Section 3.2). The reason for this choice is twofold:

1. the family-based strategy is only possible once the intermediate result of a family-product-based analysis (the family-based step) is available; and

2. all of our feature-based strategies (left half of Figure 3.7) rely on concepts that are needed for the family-based ones (right half of Figure 3.7).

Additionally, we used a top-down approach, starting from a strategy and then specifying the related concepts as needed. For instance, the product-based analysis of annotative models makes use of annotative models themselves; these, in turn, depend on defining annotative PMCs, which depend on PMCs, expressions, and so on. With this approach, we expected the resulting specification to be as parsimonious as possible. This is the inverse of the presentation sequence in each subsection of Section 3.2, where concepts are presented in a bottom-up fashion as building blocks from which more complex definitions are constructed.

### Choice of Proof Assistant

We started the specification work with a "dry-run", whereby we specified part of the upper right quadrant of Figure 3.7 using two proof assistants: PVS [68] and Coq [11]. The goal was to experience both tools in our concrete setting, to decide on which would be more helpful. This proof of concept, which consists of specification and proofs up to Lemma 2 (Evaluation well-definedness for annotative models), is available at `https://github.com/thiagomael/proof-assistants-poc`.

At the time of this proof of concept, we used introductory documentation and tutorials. Based on that material, we did not find a way to specify type constraints in Coq, especially for return-type predicates. For that reason, we were not capable of using sets as types for function parameters, hindering the specification of partial functions (which are needed for expression evaluations, for instance). Contrastingly, Lemma 2 was automatically generated by PVS as a type-correctness condition, based on our use of dependent

[3]Using as reference a dual core (4 threads) i7-4500U CPU with 1.8 GHz clock and 8 GB RAM running Ubuntu 19.04.

typing as a specification technique. Moreover, our Coq proofs became considerably larger than the corresponding ones in PVS (Figure 4.1).

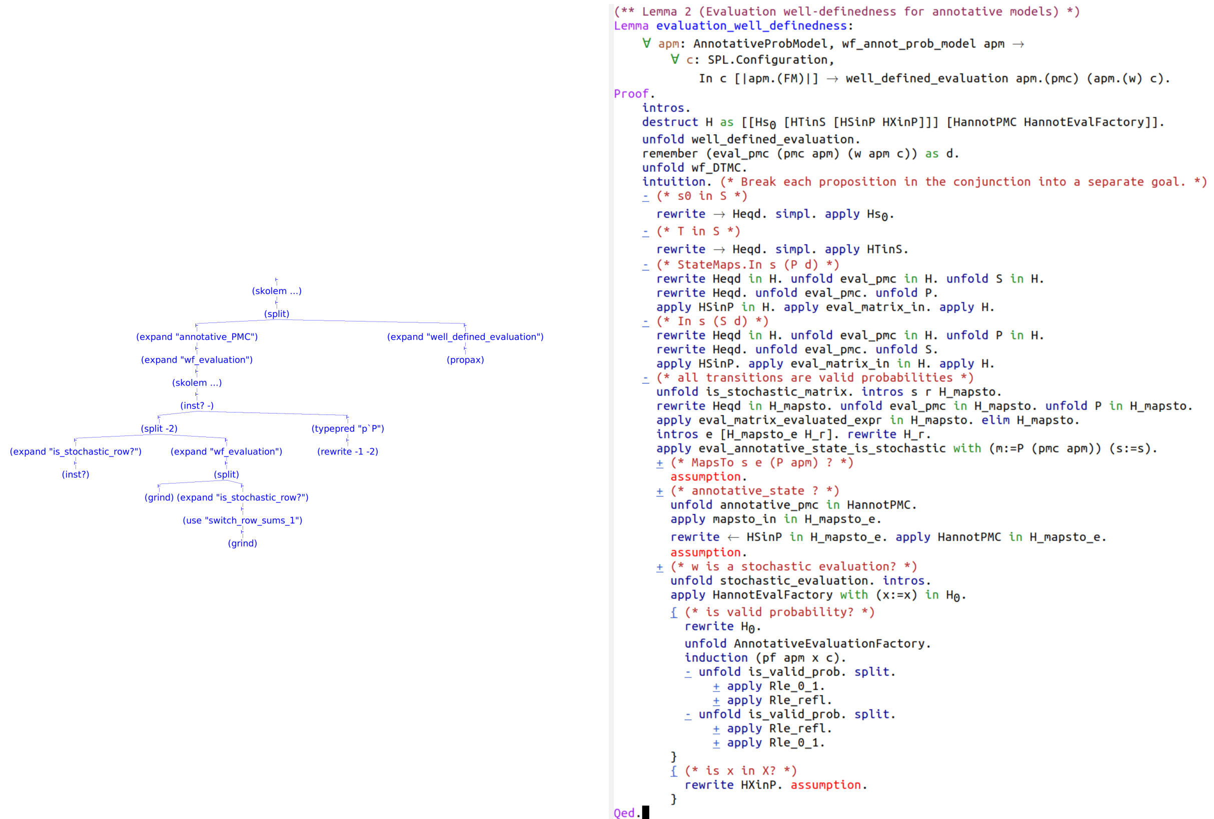



Figure 4.1: Proof of Lemma 2 in PVS (left) and Coq (right)

Also, the PVS type checker identified a corner case of our specification: we did not account for the possibility of divide-by-zero errors. Coq, on the other hand, does not restrict the division operator to non-zero denominators.[4] This capability of the PVS type checker led us to believe that this tool would be more helpful to detect specification mistakes. One disadvantage of using PVS, however, is that Coq provides the ability to extract certified Haskell code from a working specification—a feature that could be helpful in future work, as a means to derive a certified product-line reliability analysis tool.

Note, however, that we did not perform a thorough and systematic comparative investigation. Rather, we sought for an educated guess about what capabilities each tool had to offer. Indeed, both PVS and Coq allowed us to specify and prove our theory up to the same point.

[4]This behavior is a sound design decision to simplify Coq specifications. Hence, division by zero (if relevant to the specification at hand) must be conciously introduced as a premise.

### Design Principles

During the mechanization process, we followed a number of guidelines to help us specify a sound and readable theory.

**Operational specification style:** The original theory started as a formalization of techniques implemented in a tool designed to empirically compare user-oriented reliability analysis strategies [54]. Aiming to eventually close the loop and implement a tool based on the mechanized theory, we favored an *operational* specification style (i.e., concrete definitions of functions and data structures), as opposed to a *declarative* style (i.e., uninterpreted definitions that are defined in terms of their desired properties).

Moreover, some results that we use (e.g., concerning DTMCs) are not mechanized in PVS. To be able to specify our analysis strategies and prove their soundness, we created PVS theories for these results. In contrast with our own work, however, we assume that these third-party results are correct. Thus, we only specify their properties when (and if) they are needed—what could be called a "lazy specification" strategy. Also, when specifying such properties, we employ a declarative style to abstract their inner workings.

**Tuples as records:** Whenever an element of our theory is defined as a tuple (e.g., Definition 1 – Parametric Markov Chain), we specify this element in PVS using a record type. Records provide named accessors, instead of the positional accessors available in tuple types. This makes the specification easier to read and to maintain.

**Partial functions with predicate subtypes:** PVS only supports total functions. To implement partial functions in PVS, we must either restrict the domain to a set of valid inputs, or specify some notion of invalid output to be returned whenever the function is not defined [67]. We chose the former approach, that is, to limit the domain by means of predicate subtypes. This way, we can leverage the type checker to enforce constraints on input and avoid including additional restrictions on lemmas, theorems, and definitions.

**Dependent typing:** We use dependent typing to encode constraints over data and function parameters. The rationale is the same as for predicate subtypes: since dependent types embed restrictions within the type system, the type checker is able to generate type-correctness conditions that must be proved.

**Sets as types:** Many definitions and operations within our theory rely on elements of given sets. Examples are evaluation functions (defined over a specific set of variables) and composition of PMCs (constrained to the PMC's set of states). Thus, we exploit

PVS's ability to define types from sets—which is possible because PVS represents sets as predicates over a type. This can also be seen as a special case of predicate subtypes.

**Existing libraries:** Whenever possible, we leverage existing libraries to model auxiliary concepts. In particular, we exploit a number of theories in the NASA PVS Library[5] (cf. Appendix C, Figure C.1). Given that this library is used in formal verification of mission-critical software, we assume that the risk of introducing inconsistencies is low.

## 4.2 Walk-through

In this section, we present the process of mechanizing our manually-specified theory (Chapter 3), focusing on the most relevant aspects and illustrating the design decisions. We begin by laying out the fundamental definitions upon which the theory is built (Section 4.2.1), then describe the construction of family-product-based and family-based strategies (Sections 4.2.2 and 4.2.3). After that description, we discuss PMC composition and the feature-product-based and feature-family-based strategies (Section 4.2.4). Last, we bridge the two sides of our commuting diagram with a discussion on the mechanization of variability encoding (Section 4.2.5).

### 4.2.1 Foundations

To mechanize our reliability analysis theory, we first need definitions and facts about the underlying models and the corresponding operations. The most fundamental concepts on which our theory relies are the ones related to discrete-time Markov chains and their parametric counterpart. Accordingly, we searched for existing libraries that implemented these concepts in PVS, but had no success.

A possible alternative would be to produce a new library building on definitions from the `sigma_set` and `probability` libraries (distributed along the NASA libraries package) and following the formalization presented by Baier and Katoen [7]. However, since the soundness of facts concerning DTMCs and PMCs is not the focus of this work, a formal specification of these concepts is out of scope. Thus, we made a design choice to specify only as much as needed, using an axiomatic style to abstract details that are not directly needed to the specification of commuting strategies.

For instance, since our definitions rely on the manipulation of states and transition matrices of Markov chains, we define a DTMC as a record type with these elements as components:

[5] https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/

```
1  DTMC: TYPE =
2      [# S : non_empty_finite_set[state],
3         s0: (S),
4         P : {m: real_transition_matrices.transition_matrix | dom(m) = S},
5         T : non_empty_finite_set[(S)] #]
```

In the above definition, we fix a finite set `S` of states (according to the assumptions we made for the reliability models) and establish that the initial state $\mathtt{s}_0$ and the set `T` of target states are all taken from `S`—hence the use of `(S)`, meaning the type of states $s$ such that $s \in$ `S`. Furthermore, the transition matrix `P` is such that its domain is exactly `S` (the image is the set of probabilities—i.e., the Real interval $[0, 1]$).

We defined the type `state` as an alias to PVS's own `nat`. The reason for not leaving `state` uninterpreted is that we need the set of possible states to be infinite in order to allow arbitrary renamings (see Definition 33). Although we could provide an axiomatic definition for an infinite state type, this would inevitably mimic the one already provided in the NASA library `sets_aux@infinite_nat_def`. Hence, it was simpler to just label states as Natural numbers, without loss of generality.

To avoid unneeded detail, our theory does not rely on any specific algorithm for probabilistic model checking. Therefore, the probability of reaching the set of success states in a DTMC ($Pr^{\mathcal{D}}(s, T)$ in Property 1 – Reachability probability for DTMCs) is left as an uninterpreted function type:

```
1  prob_set(d: DTMC, s: (d`S), T: non_empty_finite_set[(d`S)]): probability
2
3  % Definition 20 - Non-parametric model checking
4  alpha(d: DTMC): probability = prob_set(d, d`s0, d`T)
```

Properties of these models, such as Property 1 (Reachability probability for DTMCs), are presented as axioms stated over the definitions we provide. These properties are assumed from the results in the corresponding literature (in this particular case, the book by Baier and Katoen [7]), to abstract details that are not directly needed in our theory (cf. Section 4.2.1).

```
1  % Auxiliary definition - Reachability in a DTMC
2  reachable?(d: DTMC, s1, s2: (d`S)): INDUCTIVE boolean =
3    s1 = s2
4    OR
5    EXISTS (s3: (successors(d, s1))): reachable?(d, s3, s2)
6
7  reachable?(d: DTMC, s: (d`S), T: non_empty_finite_set[(d`S)]): boolean =
8    EXISTS (ss: (T)): reachable?(d, s, ss)
```

```
% Property 1 - Reachability probability for DTMCs
reachability_probability_property: AXIOM
  FORALL (d: DTMC, s: (d`S), T: non_empty_finite_set[(d`S)]):
    prob_set(d, s, T) = COND
      member(s, T) -> 1,
      NOT reachable?(d, s, T) -> 0,
      ELSE -> sum[(d`S), real, 0, +](d`S,
                                       LAMBDA (ss: (d`S)):
                                         (trans(trans(d`P)(s))(ss) *
                                            prob_set(d, ss, T)))
    ENDCOND
```

Note that we needed to provide an explicit inductive definition for DTMC reachability in Line 2, whereby `reachable?`$(\mathtt{D}, \mathtt{s}_1, \mathtt{s}_2)$ is true iff $s_1 \rightsquigarrow s_2$. Also, the auxiliary function `sum` in Line 16 is defined in the PVS library `finite_sets_sum`. This function is such that `sum(S, f)` has the same semantics as $\sum_{s \in S} f(s)$.

In the definitions so far, we use the predicate subtype `(d‘S)` to enforce that states belong to the set `S` of the DTMC record `d`. We also introduced the `dom` and `trans` functions, which are accessors for the record types that denote transition matrices and their corresponding rows. The `dom` accessor yields the matrix domain, whereas `trans` gives us the actual transition function (i.e., a function that returns the transition probability for a given pair of states).

```
transition_matrices [V: TYPE] : THEORY
BEGIN
  IMPORTING states,
            finite_sets[state]

  transition_row: TYPE =
    [# dom  : finite_set[state],
       trans: [(dom) -> V] #]
  transition_matrix: TYPE =
    [# dom  : finite_set[state],
       trans: [(dom) -> {r: transition_row | r`dom = dom}] #]

  image(r: transition_row): finite_set[V] =
      image(r`trans, r`dom)

  image(m: transition_matrix): finite_set[V] =
      {v: V | EXISTS(s1: (m`dom)): image(m`trans(s1))(v)}

END transition_matrices
```

An alternative representation would be to define rows and matrices as functions:

```
alt_transition_matrices [S, V: TYPE] : THEORY
BEGIN
  transition_row: TYPE = [S -> V]
  transition_matrix: TYPE = [S -> transition_row]
END alt_transition_matrices
```

Defining transition matrices as records has the goal to make it easier to reference their domain. It is an open issue to compare these alternative representations with regard to the complexity of generated proof obligations.

In the PVS theory for transition matrices, we defined the image of a transition matrix (i.e., the set of all transition values for some pair of states) in a declarative way, using set comprehension notation. This is not contrary to our design principle of favoring operational specifications (Section 4.1), since this particular definition is only used to specify the semantics of transition matrices, and not to state the behavior of analysis strategies.

Moreover, we made the theory `transition_matrices` parametric on the type of transition labels, since we must represent both Real transition matrices and parametric ones. This theory is further specialized in `real_transition_matrices` and `parametric_transition_matrices`. The former is mostly an instance of the theory `transition_matrices[probability]`; the latter, on the other hand, is more involved, since it deals with facts about rational expressions. Also, to conform with our specification of DTMC, we used finite sets of states for transition matrices.

### An Interlude on Rational Expressions

To explain parametric transition matrices and PMCs, we must first cover another fundamental concept in our work: rational expressions. The manual version of our theory establishes semantics and notation for expression evaluation (Definition 2), but we did not elaborate on this subject because it is a fairly intuitive notion to a human reader. PVS, on the other hand, must be given precise definitions.

The NASA library for Bernstein polynomials provides a specification of multi-variate polynomials, which could be used to model rational expressions. However, this third-party library represents polynomials as sequences of Real coefficients, whereas we are interested in manipulating variables as syntactic objects. So, we specified a theory of rational expressions and expression evaluation (rational_expressions):

```
variable: TYPE

rat_expr: DATATYPE
  BEGIN
```

```
5        const(r: real): const?
6        variable(x: variable): variable?
7        minus(a: rat_expr): minus?
8        sum(a,b: rat_expr): sum?
9        sub(a,b: rat_expr): sub?
10       mul(a,b: rat_expr): mul?
11       div(a,b: rat_expr): div?
12       exp(a: rat_expr, i: nat): exp?
13   END rat_expr
```

An expression is defined as an abstract datatype whose structure mimics the supported algebraic operations (Line 3).[6] In this definition, we consider variables to be members of an uninterpreted type, since we are not interested in the syntactic aspects of expressions. We then specify how to obtain the set `X` of variables in an expression as a function `vars: [rat_expr -> finite_set[variable]]`, defined recursively on the structure of the expression. This is an algorithmic way to obtain the set $X$ of variables such that an expression $\varepsilon$ belongs to $\mathcal{F}_X$ (Section 2.2.1).

An evaluation function follows the original definition as a mapping from a given set of variables to Real numbers. To approach an operational definition, we constrain the domain of an evaluation to be a *finite* set—that is, we return to the definition of evaluations as partial functions, given by Hahn et al. [41] (cf. Section 2.2.1). The actual evaluation of an expression `e` is then defined as a function that takes `e` and an evaluation function for `vars(e)` as parameters and recursively performs arithmetics on `e`'s structure.

```
1  evaluation(X: finite_set[variable]): TYPE = [(X) -> real]
2
3  % Definition 2 - Expression evaluation
4  eval(e: rat_expr, u: evaluation(vars(e))): RECURSIVE maybe_real =
5    CASES e of
6        const(r)     : a_real(r),  % Lifts r to a maybe_real
7        variable(x) : a_real(u(x)),
8        minus(a)     : - eval(a, u),
9        sum(a, b)    : eval(a, restrict(u)) + eval(b, restrict(u)),
10       sub(a, b)    : eval(a, restrict(u)) - eval(b, restrict(u)),
11       mul(a, b)    : eval(a, restrict(u)) * eval(b, restrict(u)),
12       div(a, b)    : eval(a, restrict(u)) / eval(b, restrict(u)),
13       exp(a, i)    : eval(a, u) ^ i
14   ENDCASES
15   MEASURE e BY <<  % The termination measure "«" is the well-founded order
16                    % that PVS automatically generates for an ADT
17
```

[6]Since we are dealing with *rational* expressions, we only model the operations needed by fractions of polynomials—basic arithmetics and exponentiation to Natural powers.

```
18 real_evaluation?(e: rat_expr)(u: evaluation(vars(e))): boolean =
     is_real?(eval(e, u))
```

When defining `eval` (Line 4), we had to specify a concept that was implicit in the original version of our theory: expression evaluation does not yield Real values in general. In the presence of the division operation (which is possible, since a rational expression is a fraction of polynomials), there can be evaluations that cause a given expression to divide by zero. Thus, we specified that `eval` returns a value of type `maybe_real`, which can be either a Real value or the constant `undefined` (see Section 2.4). Accordingly, we defined a predicate over evaluation functions to be able to specify whether they evaluate an expression to a valid Real number (Line 18). This is the foundation on which we later build the concept of well-defined (or *well-formed*) evaluation (Definition 3).

At this point, it is worth noting that the definition of `eval` uses the function `restrict` from the PVS prelude—which is extensively used throughout our mechanization. This function takes a function `[S -> R]` and *restricts* its domain `S` to a subtype `T` of `S`. In this particular case, the imposed constraint turns an evaluation over `vars(e)` into an evaluation operating only on `vars(a)` or `vars(b)` (for subexpressions `a` or `b`, which the type checker is able to infer from the context). Without the calls to `restrict`, type checking fails because `eval`'s signature uses dependent types to constrain that we only allow the evaluation of all variables of an expression.

Another detail that was overlooked in the manual specification—but exposed by the mechanization effort—is that we implicitly considered constant expressions (i.e., the ones with an empty set of variables) as Reals. To overcome this issue, we specified an inductive predicate `const_expr?` that is true for constant expressions, along with a function that explicitly converts such constants to the corresponding Real numbers. This conversion relies on the fact that, since a constant expression does not have variables, it can be evaluated to the same value by using any evaluation function.

```
1 real_const_expr?(e: (const_expr?)): boolean =
2     FORALL (X: finite_set[variable], u: evaluation(X)):
3         is_real?(eval(e, restrict(u)))
4 const_expr_to_real(e: (real_const_expr?)): real =
5     eval(e, LAMBDA (x:variable): 0)
6 % The next line tells PVS to apply the conversion function whenever needed.
7 CONVERSION+ const_expr_to_real
```

### Parametric Markov Chains

Having handled the formerly implicit concepts regarding rational expressions, we return to our discussion of parametric transition matrices. The corresponding theory required a more involved specification because it covers the evaluation of parametric matrices to

yield regular (Real) ones. Hence, we had to provide predicates and judgements to handle type checking issues that arise both from the possibility of divisions by zero and from the need to convert degenerate parametric matrices that only have constant expressions as transition labels.

For instance, we define the evaluation of a row in a parametric transition matrix by mapping the expression evaluation over each transition in the given row:

```
parametric_transition_row: TYPE = transition_row[rat_expr]
evaluation(r: parametric_transition_row): TYPE = evaluation(vars(r))
% First item of Definition 3 - Well-defined evaluation
pre_wf_evaluation(r: parametric_transition_row)
                 (u: evaluation(r))
        : boolean =
    FORALL (e: (image(r))):
        LET v = eval(e, restrict(u)) IN
          is_real?(v) AND num(v) >= 0 AND num(v) <= 1

mapped_real_transition_row(r: parametric_transition_row): TYPE =
    {rr: real_transition_row | dom(rr) = dom(r)}

eval(r: parametric_transition_row, u: (pre_wf_evaluation(r)))
        : mapped_real_transition_row(r) =
    map(r, LAMBDA (e: (image(r))): num(eval(e, restrict(u))))
```

Note that this definition establishes that the result is a Real transition row with the same domain (i.e., the set of states is not changed). However, since the `eval` function for rational expressions returns a `maybe_real`, the type checker generates proof obligations (TCCs) requiring that we show, for this particular case, that the results are not `undefined`. To avoid the generation of such TCCs for all occurrences of `eval`, we prove these facts once in the form of judgements, which then become available to the type checker:

```
evaluated_expr_is_real: JUDGEMENT
    FORALL(r: parametric_transition_row,
           u: (pre_wf_evaluation(r)),
           e: (image(r))):
        eval(e, restrict(u)) HAS_TYPE (is_real?)

evaluated_expr_is_probability: JUDGEMENT
    FORALL(r: parametric_transition_row,
           u: (pre_wf_evaluation(r)),
           e: (image(r))):
        num(eval(e, restrict(u))) HAS_TYPE probability
```

Similar to the theory of rational expressions, our theory of parametric transition matrices specifies the predicates `const_row?` and `const_matrix?` (which are true for rows and matrices whose transition labels are all constant expressions) and appropriate conversions to Real rows and matrices.[7] Moreover, this theory also defines what it means for an evaluation to be well-defined for transition rows and matrices:

```
wf_evaluation(r: parametric_transition_row)
             (u: evaluation(r))
        : boolean =
    pre_wf_evaluation(r)(u) AND is_stochastic_row?(eval(r, u))
% Overloading of the predicate for rows
wf_evaluation(m: parametric_transition_matrix)
             (u: evaluation(m)): boolean     =
    FORALL (s: (dom(m))):
        wf_evaluation(trans(m)(s))(restrict(u))
```

With these definitions, we are now able to specify PMCs and how to convert them to DTMCs:

```
% Definition 1 - Parametric Markov Chain
PMC: TYPE = [# S : non_empty_finite_set[state],
               s0: (S),
               P : {m: parametric_transition_matrix | dom(m) = S},
               X : {V: finite_set[variable] | V = vars(P)},
               T : non_empty_finite_set[(S)] #]

% Definition 3 - Well-defined evaluation
well_defined_evaluation(p: PMC)(u: evaluation(p`P)): boolean =
   wf_evaluation(p`P)(u)

eval(p: PMC, u: (well_defined_evaluation(p))): DTMC =
    (# S  := p`S,
       s0 := p`s0,
       P  := eval(p`P, u),
       T  := p`T #)
```

Properties of these models, such as Lemma 1 (Parametric probabilistic reachability soundness), are presented as axioms stated over the definitions we provide. Their proof is assumed from the results in the corresponding literature (in this particular case, the work by Hahn et al. [41]), to abstract details that are not directly needed in our theory (cf. Section 4.2.1).

```
% Definition 21 - Parametric model checking
```

[7]These predicates also encode the stochastic property—i.e., every row must sum up to 1.

```
2  alpha_v(p: PMC): {e: rat_expr | vars(e) = p`X}  % uninterpreted
3  % A fact implied in the paper by Hahn et al. [41]:
4  alpha_v_eval_is_real: AXIOM
5      FORALL (p: PMC, u: (well_defined_evaluation(p))):
6          is_real?(eval(alpha_v(p), u))
7
8  % Lemma 1 - Parametric probabilistic reachability soundness
9  parametric_reachability_soundness: AXIOM
10     FORALL (p: PMC, u: (well_defined_evaluation(p))):
11         prob_set(eval(p, u), p`s0, p`T) = num(eval(alpha_v(p), u))
```

With this axiomatic definition, we have specified both parametric ($\hat{\alpha}$, denoted by `alpha_v` in PVS) and non-parametric ($\alpha$) model checking functions. This is the foundation upon which we state our first commutativity lemma:

```
1  % Lemma 3 - Commutativity of PMC and expression evaluations
2  eval_commutativity: LEMMA
3      FORALL (p: PMC, u: (well_defined_evaluation(p))):
4          alpha(eval(p, u)) = eval(alpha_v(p), u)
```

Finally, we also declare a predicate and a conversion to deal with constant PMCs—i.e., the ones with constant transition matrices, which can be trivially represented as DTMCs. This way, we can formally apply $\alpha$ to PMCs (as we informally did in Section 3.2) by using an automatic PVS conversion that only works if the PMC at hand is really constant. This fills a gap in the hand-made specifications.

### Software Product Lines

Before we can talk about the specification of analysis strategies, it is important to mention how we model SPL-related concepts. As with Markov chains, we chose a declarative style of specification for product lines, since our focus is the analysis of implementation assets.

```
1  SPL : THEORY
2  BEGIN
3    FM: TYPE+                                    % Feature Models
4    name: TYPE+                                  % Names of features
5    configuration: TYPE = finite_set[name]       % Selection of features
6
7    % Set of features in the feature model
8    features(fm: FM): non_empty_finite_set[name]
9    % Cardinality of the set of features
10   ; ##(fm: FM): posnat = card(features(fm))
11   % FM semantics - [[FM]]
12   [||](fm: FM): {C: non_empty_finite_set[configuration] |
13                  FORALL (c: (C), f: (c)): features(fm)(f)}
14 END SPL
```

The previous listing shows our theory of software product lines in its entirety. The only properties of a product line that we actually need are that (a) it has a finite set of features (Line 8) and (b) it has a feature model that is consistent and has a finite set of possible configurations as its semantics (Lines 12 and 13). We only handle finite feature sets because the presence of each feature in a given configuration must be mapped to a variable in an ADD (which is a finite data structure), which is also why we need the cardinality of the feature set (Line 10). Accordingly, the set of possible configurations must also be finite.[8] Moreover, since we do not manipulate feature expressions nor feature model constraints, such concepts may be left unspecified.

However, leaving the feature model semantics and the function `features` unspecified leads to existence TCCs. For instance, the following listing shows the obligation to prove that there is at least one function that maps a feature model into a non-empty set of feature names:

```
1 % Existence TCC generated (at line 9, column 2) for
2     % features(fm:  FM): non_empty_finite_set[name]
3   % unfinished
4 features_TCC1: OBLIGATION
5   EXISTS (x: [FM -> non_empty_finite_set[name]]): TRUE;
```

For that reason, we declare `FM` and `name` as non-empty types, using the keyword `TYPE+`. Thus, we are able to provide a trivial witness for that obligation: `LAMBDA (fm: FM): singleton(choose({n: name | TRUE}))` (i.e., we pick an arbitrary feature name, which requires `name` to be non-empty).

We note that there are third-party PVS theories of software product lines available for reuse [81]. However, these theories specify facts about product lines themselves, such as well-formedness of feature expressions and validity of configurations. Our work, on the other hand, abstracts such details of product line assets. Thus, importing existing SPL theories would introduce an external dependency (along with the corresponding threats) without perceived benefits.

## 4.2.2 Family-product-based Strategy

After establishing the foundations, we start the mechanized specification with the upper right quadrant of Figure 3.7.

[8] For scoping reasons, we do not support cardinality-based feature models [28], which would allow features to "repeat" within a given configuration. Hence, a finite set of features implies that the configuration space must also be finite.

### Annotative Models

To define annotative models (Definition 8), we first define annotative PMCs by using a predicate over the type `PMC`:

```
% Definition 5 - Annotative PMC
annotative_PMC(p: PMC): boolean =
    FORALL (s: (p`S)):
        const_row?(trans(p`P)(s))          % First item
      OR
        is_switch_row?(trans(p`P)(s), s)  % Second item
```

The predicate `is_switch_row?` is true for states that act as behavior switches—i.e., states that fall in Item 2 of Definition 5:

```
% A row in a parametric transition matrix such that there
% are only "switch" transitions (with probabilities x and 1-x).
is_switch_row?(r: parametric_transition_row, s: (dom(r))): boolean =
    EXISTS (s1, s2: (dom(r)), x: variable):
        s1 /= s2 AND s /= s1 AND s /= s2
      AND
        trans(r)(s1) = variable(x)
      AND
        trans(r)(s2) = sub(const(1), variable(x))
      AND
        FORALL (s3: (remove(s2, remove(s1, dom(r))))):
            trans(r)(s3) = const(0)
```

Note that Line 5 assures that neither of the switch transitions is a loop. This constraint is needed to prove some facts about slots and variability encoding later on, but this is not part of the hand-made Definition 5 (Annotative PMC). To fix this issue, the corresponding item of that definition should read as follows:

$$\exists_{s_0, aft_s \in \underbrace{S \setminus s}_{\text{this was missing}}} \exists_{x \in X} \cdot Succ(s) = \{s_0, aft_s\} \wedge \mathbf{P}(s, s_0) = x \wedge \mathbf{P}(s, aft_s) = 1 - x$$

Thus, we have another example of a gap in the original specification that was detected with the help of PVS.

The translation of other definitions needed to specify annotative models (e.g., presence functions and $\pi$) is straightforward. Nonetheless, we found it useful to break Definition 7 (Evaluation factory) in two:

```
pre_evaluation_factory(X: finite_set[variable]): TYPE =
    [([|fm|]) -> evaluation(X)]
% Definition 7 - Evaluation factory
evaluation_factory(p: PMC): TYPE =
```

```
5     {w: pre_evaluation_factory(p`X) | FORALL (c: ([|fm|])):
        well_defined_evaluation(p)(w(c))}
```

This change was due to the fact that `evaluation_factory` (Line 4) is more appropriate to use in some function definitions (e.g., $\pi$), since it encodes the constraint that the produced evaluations must be well-defined for the given PMC. This way, Lemma 2 (Evaluation well-definedness for annotative models) turned into a proof obligation (TCC). On the other hand, we cannot use `evaluation_factory` directly when defining data types or some predicates. To do this, it would be necessary that all evaluation factories yielded evaluations that are well-defined for all PMCs, which is not true. Thus, we use the "intermediate" type `pre_evaluation_factory` (Line 1) whenever we need generalized mappings from configurations to evaluation functions.

Annotative probabilistic models (Definition 8) are defined as record types, as usual. However, the feature model is not part of this record, but rather a theory parameter. The reason is that many function and type parameters are dependently-typed on the actual feature model, so it cannot be a regular PVS variable.

### Product-based and Family-product-based Analyses

To instantiate the feature model in theory parameters, we define a constant `fm: FM`[9] in our top-level theory, called `rome` because of our *All roads lead to Rome* paper [15]. This theory, besides being the entry point to our mechanized specification, is the place where our reliability analysis strategies are defined and proved sound.

Building on the specifications we have so far, we define the annotative product-based and family-product-based strategies, as well as the theorem stating their equivalence.

```
1  fm: FM  % Constant representing a given feature model FM
2  IMPORTING annotative_reliability_models[fm]
3
4  c: VAR ([|fm|])  % a variable ranging over the possible configurations
5  m: VAR annotative_reliability_model
6  % Strategy 1 - Product-based analysis of annotative models
7  product_based_analysis(m, c): probability =
8      alpha(pi(m`P, m`w, c))
9
10 % Strategy 3 - Family-product-based analysis
11 family_product_based_analysis(m, c): probability =
12     sigma(alpha_v(m`P), m`w, c)
13
14 % Theorem 1 - Soundness of family-product-based analysis
15 family_product_soundness: THEOREM
```

[9] In this definition, `FM` is the type of feature models defined in the SPL theory.

```
16    FORALL (m, c):
17       family_product_based_analysis(m, c) = product_based_analysis(m, c)
```

### 4.2.3 Family-based Strategy

Following the specification strategy, we proceed to the lower right quadrant of Figure 3.7. However, at this point we need to take a step back and provide a specification for the remaining foundational data structure—ADDs.

#### Algebraic Decision Diagrams

As with Markov chains, we searched for reusable third-party libraries, to no avail. Therefore, we had to provide our own specification of ADDs. According to our design principles, we followed a declarative style for this data structure, since it is not our focus to prove facts about ADDs. Future work may seek to instantiate our ADD theories with operational specifications.

Our declarative specification views an ADD as an $n$-ary function of Boolean parameters to a type `T` over which we have operations of interest.

```
1  ADD_def [n: posnat] : THEORY
2  BEGIN
3    IMPORTING structures@arrays[n]     % NASA library
4    variables: TYPE = ArrayOf[boolean]
5    ADD[T: TYPE]: TYPE = [variables -> T]
6  END ADD_def
```

The actual operations on ADDs are then defined based on their denotational semantics:

```
1  ADD_ops [T: TYPE, n: posnat] : THEORY
2  BEGIN
3    IMPORTING ADD_def[n]
4    val: VAR variables
5    % Lifting of codomain value into constant ADD
6    constant(t: T): ADD[T] = LAMBDA(val): t
7    CONVERSION constant
8
9    f, g: VAR ADD[n][T]
10   op: VAR [T, T -> T]
11   unary_op: VAR [T -> T]
12
13   % ADD operations
14   apply(f, g, op): ADD[T] = LAMBDA val: op(f(val), g(val))
15   unary_apply(f, unary_op): ADD[T] = LAMBDA val: unary_op(f(val))
16
```

```
17    test: VAR ADD[boolean]
18    % if-then-else operator
19    ite(test, f, g): ADD[T] = LAMBDA val:
20        IF test(val)
21            THEN f(val)
22            ELSE g(val)
23        ENDIF
24  END ADD_ops
```

Note that, although we provide actual definitions for ADD operations (Lines 6, 14, 15 and 19), these definitions do not correspond to the algorithms that make ADDs suitable for efficient computation with Boolean functions. They are just operational definitions of the functions that denote their semantics.

Finally, since we use ADDs to denote functions from configurations to reliability values, we need a way to unambiguously map product-line configurations to ADD variables (cf. last paragraph of Section 2.1.1), such that any given variable always corresponds to the presence (or absence) of the same feature. In the following, we do this by first creating a list with all members of the feature set (Line 5), effectively defining an arbitrary (but fixed) ordering among features. This is equivalent to indexing the features as $\{F_1, F_2, ..., F_n\}$. Then we define an injective mapping from configurations over $n$ features to $n$-ary Boolean arrays (`to_ADD_variables`). Conversely, we can invert this injection to recover the configuration that maps to a given instantiation of ADD variables (`to_conf`).

```
1  IMPORTING ADD_def[ ##(fm)],
2            structures@set2seq[SPL.name]
3
4  % Arbitrary (but fixed) feature order
5  feature_order: finite_sequence[SPL.name] = set2seq(features(fm))
6
7  % An injective function mapping a feature selection to ADD arguments
8  to_ADD_variables(c: ([|fm|])): ADD_def.variables =
9      LAMBDA (i: below( ##(fm))):
10         member(feature_order(i), c)
11 % ... and a way to recover the original selection
12 to_conf(val: variables): ([|fm|]) = inverse(to_ADD_variables)(val)
```

In this listing, `inverse` (Line 12) is a higher-order function from the PVS prelude that takes a function $f$ and produces a function that, given a value $y$, returns $x$ such that $f(x) = y$. Moreover, we use the cardinality of the set of features (`##(fm)`) to instantiate the `ADD_def` theory, so that the input of ADDs consists of exactly one parameter for each available feature.

### Expression Lifting

We use ADDs to perform efficient computations over lifted expressions. In the manual version of our theory, we defined expression lifting as a semantic change, so that expressions remained the same objects after lifting. Since our PVS specification of expression semantics is given by the function `eval`, we provide an ADD-based evaluation semantics for lifted expressions:

```
% n is a theory parameter of type nat
ADD_evaluation(X: finite_set[variable]): TYPE = [(X) -> ADD[n][real]]

% Definition 23 - Expression lifting
eval (e: rat_expr, u: ADD_evaluation(vars(e))): RECURSIVE
    ADD[n][maybe_real] =
  CASES e of
      const(r)    : real_constant(r),
      variable(x) : to_maybe(u(x)),
      % The remaining cases are omitted for brevity
  ENDCASES
  MEASURE e BY <<
```

Thus, we do not define a mechanized `lift` operator for expressions; instead, we use the overloaded `eval` function and define a predicate to assert whether an `ADD_evaluation` is a lifted counterpart to a given evaluation factory.

```
lifted_evaluation_factory(X: finite_set[variable]): TYPE =
    ADD_evaluation(X)

% Definition 24 - Lifted evaluation factory
lifted(X: finite_set[variable], w: pre_evaluation_factory(X))
      (lw: lifted_evaluation_factory(X))
        : boolean =
     FORALL (x: (X), val: variables):
        lw(x)(val) = w(to_conf(val))(x)
```

Building on the definitions and lemmas so far, we are able to define the lifted annotative evaluation factory—$\hat{p}$ in Lemma 5 (Soundness of lifted annotative evaluation factory). Then, we proceed to define the family-based analysis strategy and state its soundness.

```
lifted_annotative_evaluation_factory(X: finite_set[variable], pf:
    presence_function(X))
        : lifted_evaluation_factory(X) =
     LAMBDA (x: (X)):
        LAMBDA (c: ([|fm|])):
            IF pf(x, c) THEN 1
            ELSE 0
```

```
            ENDIF

% Strategy 4 - Family-based analysis
family_based_analysis(m): ADD[maybe_real] =
    sigma_v(alpha_v(m`P), lp)
    WHERE lp = lifted_annotative_evaluation_factory(m`P`X, m`pf)

% Theorem 4 - Soundness of family-based analysis
family_based_analysis_soundness: THEOREM
      FORALL (m, c):
          family_based_analysis(m)(c) = product_based_analysis(m, c)
```

### 4.2.4 Feature-based Strategies

Since our notion of compositional models relies on our definitions of annotative models, we define the upper left quadrant of Figure 3.7 by leveraging the mechanized specification produced so far.

#### Compositional PMC

From the manual version of our theory, we know that a compositional PMC is a special case of annotative PMC (see Definition 10). However, we have no documented way of extending the record type PMC. Thus, we declare a new record type for compositional PMC data (Line 2), along with a conversion to the regular PMC record type (Line 11) and a predicate that reuses the predicate for annotative PMCs (Line 18):

```
% Definition 10 - Compositional PMC (tuple definition)
compositional_PMC_data: TYPE =
    [# S    : non_empty_finite_set[state],
       s0   : (S),
       s_suc: (S),
       s_err: (S),
       P    : {m: parametric_transition_matrix | dom(m) = S},
       X    : {V: finite_set[variable] | V = vars(P)},
       T    : {T: (singleton?[(S)]) | T(s_suc)} #]

pmc(p: compositional_PMC_data): PMC = (# S := p`S,
                                         s0:= p`s0,
                                         P := p`P,
                                         X := p`X,
                                         T := p`T #)

% Definition 10 - Compositional PMC (predicates)
compositional_PMC(p: compositional_PMC_data): boolean =
```

```
19      annotative_PMC(pmc(p))
20      AND p`s0 /= p`s_suc
21      AND p`s0 /= p`s_err
22      AND p`s_suc /= p`s_err
23      AND bscc(p`P) = {s: state | (s = p`s_suc) OR (s = p`s_err)}
```

In the previous definition, function `bscc` in Line 23 returns the bottom strongly-connected components of the compositional PMC. We follow our design principles and specify this function in a declarative way, since we are not interested in operationalizing an algorithm to find bottom strongly-connected components. However, we depart from the definition given by Baier and Katoen [7] and specify that a BSCC is just an absorbing state:

```
1  bscc?(m: parametric_transition_matrix)(s: (dom(m))): boolean =
2      trans(m)(s)`trans(s) = const(1)
3      AND
4      FORALL (s2: (dom(m))):
5          (s2 /= s IMPLIES trans(m)(s)`trans(s2) = const(0))
6
7  bscc(m: parametric_transition_matrix): finite_set[(dom(m))] =
8      {s: (dom(m)) | bscc?(m)(s)}
```

Every state that satisfies the predicate `bscc?` is a bottom strongly-connected component, but not every BSCC satisfies this predicate. Nonetheless, this "narrow" definition is sufficient for our purposes, since we are interested in PMCs whose only BSCCs are the two singletons induced by their success and error states.

### Dependency relation

To define a compositional probabilistic model, we first define a superset $I$ of the variables in its set $\mathscr{P}$ of PMCs and a bijection between $\mathscr{P}$ and $I$ (Definition 13 – Identifying function).

```
1  P: VAR non_empty_finite_set[(compositional_PMC)]
2  % All variables in the compositional PMCs of set P.
3  vars(P): finite_set[variable] =
4      {x: variable | EXISTS (p: (P)): vars(p)(x)}
5  % Auxiliary type denoting all strict supersets of vars(P)
6  supervars(P): TYPE =
7      {I: finite_set[variable] | card(I) = card(P)
8                                 AND strict_subset?(vars(P), I)}
9  % Definition 13 - Identifying function
10 identity_function(P, (I: supervars(P))): TYPE = (bijective?[(P), (I)])
```

The induced dependency relation $\prec$ is then defined in a theory parameterized on `P`, `I`, and `identity_function` (compositional_PMC_order). This theory also defines

predicates for minimal and maximal PMCs (Definition 15) and presents lemmas stating their existence. These definitions and lemmas come as a straightforward translation from their manual counterparts. However, the manual proof of Lemma 10 (Existence of maximal PMCs) consisted of about 6 lines of argumentation, whereas its mechanized version required 4 auxiliary lemmas (well_founded_lemmas) that make use of 3 additional theories in the NASA library (`monotone_sequences`, `finite_pointwise_orders`, and `well_foundedness`, all in the `orders` library).

### PMC Composition

The last element that is needed to define compositional reliability models is the notion of PMC composition. The original definition (Definition 34) relies on a disjoint union operator, which, to the best of our knowledge, does not have a PVS specification. Since total PMC composition also depends on the concept of PMC renaming (Definition 33), we defined renaming in a way that allowed us to produce mutually disjoint PMCs that are isomorphic to the original one. This solved the lack of a disjoint union concept and spawned a new theory of its own (PMC_renaming).

Another challenge faced when specifying PMC composition was how to operationalize the intuition of composing over all slots at once. The first idea that comes to mind in this situation is to specify how to compose PMCs over a single slot, and then recursively define a composition over the remaining (partially composed) PMC. However, this approach fails unless we take care to preserve slots that were *introduced* by previous composition steps. We actually made this mistake when we first attempted to mechanize PMC composition, but PVS helped us to identify the issue by reaching a dead-end in one of the proof branches.

Moreover, there is a considerable amount of properties that must be preserved by composition. For instance, constant transitions in the original PMCs must remain that way, and original variables must be replaced by the ones in the composed chains. Nonetheless, original variables of the base chain may still belong to the resulting set of variables; indeed, although a PMC composed over a slot for a variable $x$ must not contain $x$ itself,[10] it may contain another variable $y$ that was also in the base chain.

Thus, we specified PMC composition incrementally, so that we could state and prove lemmas regarding the preservation of required properties throughout the process. First we defined how to compose a transition matrix over a single slot of another matrix, provided that their sets of states are already disjoint. This led to the definition of how to compose two PMCs whose transition matrices satisfy those pre-conditions.

```
1 % Composes a PMC whose set of states is already disjoint with the base.
```

[10] This would violate the well-foundedness of the dependency relation.

```
2  compose_single_slot_disj(p_base: (compositional_PMC),
3                           p_comp: {p: (compositional_PMC) |
                               disjoint?(p`S, p_base`S) },
4                           sl: (slot?(p_base)))
5          : (compositional_PMC) =
6      LET P_ = compose_matrices(p_base`P,
7                                p_comp`P,
8                                sl,
9                                interface(p_comp))
10     IN
11         p_base WITH [ `S := union(p_base`S, p_comp`S),
12                       `P := P_,
13                       `X := (vars(P_)) ]
```

Then we generalized this notion to compose arbitrary PMCs, but still restricted to utilize a single slot:

```
1  % Composes an arbitrary PMC over another.
2  compose_single_slot(p_base: (compositional_PMC),
3                      p_comp: (compositional_PMC),
4                      sl: (slot?(p_base)))
5          : (compositional_PMC) =
6      LET p_comp_ = rename(p_comp, p_base`S) IN
7        compose_single_slot_disj(p_base, p_comp_, sl)
```

The complexity increases when we generalize composition to operate on more than one slot. In this case, we fix the set of slots on which to compose as a function parameter (`sls`) and consume these slots in a recursive fashion:

```
1  compose_many_slots(p_base: (compositional_PMC),
2                     p_comp: (compositional_PMC),
3                     sls: finite_set[(slot?(p_base))])
4          : RECURSIVE {p: (compositional_PMC) |
5              subset?(p_base`S, p`S)
6              % composed slots are no longer slots:
7              AND (FORALL (sl: (sls)): NOT slot?(p)(sl))
8              % non-composed slots are preserved:
9              AND (FORALL (x: (p_base`X), sl: (slot?(p_base, x))):
10                     NOT sls(sl) IMPLIES slot?(p, x)(sl))
11             % all new slots are preserved:
12             AND (NOT empty?(sls) IMPLIES
13                    FORALL (x: (p_comp`X), sl: (slot?(p_comp, x))):
14                      EXISTS (sl_: (slot_renaming?(p_comp, p_base`S, x,
                           sl))):
15                         slot?(p, x)(sl_))
16             % slots are not created out of thin air:
17             AND (FORALL (x: variable,
```

```
18                             sl: (slot?(p, x))):
19                       slot?(p_base, x)(sl)
20                        OR
21                       (EXISTS (sl_: (slot?(p_comp, x))):
22                          slot_renaming?(p_comp, p_base`S, x, sl_)(sl)))
23                AND p`s0 = p_base`s0
24                AND p`s_suc = p_base`s_suc
25                AND p`s_err = p_base`s_err
26                AND p`T = p_base`T } =
27      IF empty?(sls)
28        THEN p_base
29        ELSE compose_many_slots(compose_single_slot(p_base,
30                                                   p_comp,
31                                                   choose(sls)),
32                                 p_comp,
33                                 rest(sls))
34      ENDIF
35  MEASURE card(sls)
```

Notice the predicates in Lines 5 to 26. Each of these predicates specify a property that must hold for the resulting PMC. Our first approach was to state these properties as separate lemmas, but the definition of `compose_many_slots` uses recursion on dependently-typed variables, where the depended-upon value changes at each recursive call (`p_base` takes the partially composed PMC at Line 29). In this case, the PVS prover rule `induct` does not work, because the induction variable has free variables in it.

To overcome this issue, we stated the desired properties as predicates of the return type, so that the type checker demands that these properties have to be preserved *after each recursive call*. This way, each of the generated TCCs embed the result of the previous call as a premise. In a sense, we can interpret that these proof obligations are themselves functioning as induction principles.

The downside of this approach is that one must take care not to specify predicates that hold *at the end* of the recursion but not after intermediate steps. That is, all predicates of the return type must be recursion invariants. Otherwise, the theory type-checks, but PVS generates TCCs that are impossible to prove.

Next, we refined the notion of composition over multiple slots, by grouping slots according to their variables (`compose_many_variables`). In this step, we faced the same issue of having properties that are complex to prove if stated afterwards. Accordingly, we solved it in the same way.

The total composition of PMCs is then defined by leveraging the previous definition, whereby the fixed set of slots is given by all slots with all variables in the base PMC:

```
1  % Definition 34 - Total PMC composition
```

```
2 compose(p: (compositional_PMC), u: composition(p`X)): (composed?(p, u))
    =
3     compose_many_variables(p, p`X, u, LAMBDA (x: (p`X)): slots(p, x))
```

#### Feature-product-based Strategy

With the foundational specifications involving PMC composition, we were able to define composition factories (Definition 17), compositional probabilistic models (Definition 18), and derivation by composition (Definition 19). These definitions are close to the manual specification, so we omit them for brevity.

To specify the feature-product-based analysis strategy and prove its soundness, we also define the product-based strategy for compositional models. The rome PVS theory was updated with these definitions:

```
1  cm: VAR compositional_reliability_model
2  % Strategy 2 - Product-based analysis of compositional models
3  product_based_analysis(cm, c): probability =
4      alpha(const_to_DTMC(pi(p, restrict(cm`w), c)))
5      WHERE p = root(cm)
6
7  % Strategy 5 - Feature-product-based analysis
8  feature_product_based_analysis(cm, c): probability =
9      sigma(alpha_v(p), restrict(w), c)
10     WHERE p = root(cm),
11           w = the_compositional_evaluation_factory(cm)
```

This specification closely resembles that on the manual theory, with two visible exceptions. First, we once more rely on the PVS function `restrict` to conform dependently-typed arguments. The second is that we needed to explicitly convert the PMC resulting from derivation to a DTMC (Line 4), so that we can apply `alpha` to this result.

The interesting fact about this conversion is that it relies on a fact that was not part of the original theory: derivation by composition (Definition 19) exhausts all variables that previously existed in the base PMC, and does not add other variables. Although Definition 19 specifies that the return type of $\pi'$ is a DTMC, we did not prove this in the manual version of the theory.[11] This is a missing specification that was detected with the help of PVS.

Moreover, PVS generated an obligation to prove that the compositional evaluation factory terminates. Different from other recursive definitions in our manual specification, this one (Definition 26) did not have an accompanying termination lemma.

[11] Nonetheless, the proof of termination (Lemma 11) implicitly uses this fact in the induction hypothesis.

The soundness of the feature-product strategy is stated as expected, following the style already used for other soundness theorems:

```
% Theorem 5 - Soundness of feature-product-based analysis
feature_product_soundness: THEOREM
    FORALL (cm, c):
        feature_product_based_analysis(cm, c) =
            product_based_analysis(cm, c)
```

The mechanized proof of this theorem, similar to its manual counterpart, makes use of Corollary 1 (r-equivalence of total composition with DTMCs and evaluation). However, this corollary is a direct consequence of Lemma 13 (r-equivalence of total composition and evaluation), which, in turn, depends on the inner workings of the parametric model checking algorithm [41] and on a more detailed specification of PMCs. Thus, we decided to only state Corollary 1 and postpone its proof to a moment when we were able to specify Lemma 13.

#### Feature-family-based Strategy

The definition of our feature-family-based strategy relied on specifying a compositional counterpart to the theory of annotative_expressions_evaluation. The new theory, besides having a similar structure to the one for annotative expressions, actually reuses some results of that other theory—e.g., the result corresponding to Theorem 2 (Soundness of variability-aware expression evaluation). Other than that, the translation from the manual specification follows the rules and patterns already described in previous sections:

```
% Strategy 6 - Feature-family-based analysis
feature_family_based_analysis(cm): ADD[maybe_real] =
    sigma_v(alpha_v(p), restrict(phi))
    WHERE p = root(cm),
          phi = lifted_compositional_evaluation_factory(cm)

% Theorem 7 - Soundness of feature-family-based analysis
feature_family_soundness: THEOREM
    FORALL (cm, c):
        feature_family_based_analysis(cm)(c) =
            product_based_analysis(cm, c)
```

### 4.2.5 Variability-encoding

After specifying both the right and left parts of Figure 3.7, we must bridge them by means of variability encoding. We decided to pursue this goal before diving into the

details needed to fill the gap in the specification of Corollary 1, to prioritize the overall soundness of the analysis framework.

Moreover, we observe in the graphs of theory dependencies (Figure D.3) that the soundness of variability encoding for PMCs (Theorem 9 – Soundness of variability encoding for expressions) relies directly on Lemma 13. Since the specification of this lemma was postponed, we begun with the variability encoding of expressions.

### Encoding of Expressions

To specify variability encoding of expressions, we first extended our theory of rational_expressions to support the generalized notion of expression evaluation. The new definitions follow the pattern of the other `eval` functions, whereby we perform a recursion on the structure of expressions.

```
1  % Generalized evaluation function:  maps variables to other rational expressions
2  generalized_evaluation(X: finite_set[variable]): TYPE = [(X) ->
       rat_expr]
3
4  % Generalized evaluation (i.e., replacing variables with other expressions,
5  % instead of real values)
6  gen_eval(e: rat_expr, u: generalized_evaluation(vars(e))): RECURSIVE
       rat_expr =
7    CASES e of
8        const(r)    : e,
9        variable(x) : u(x),
10       minus(a)    : - gen_eval(a, u),
11       % Remaining cases omitted for brevity.
12   ENDCASES
13 MEASURE e BY <<
```

A significant difference, however, is that the fact that we posed as Equation (2.1) is not a definition or an immediate conclusion, as the manual specifications seem to imply. Indeed, given an expression $\varepsilon' \in \mathcal{F}_X$ and a generalized evaluation $u : X \to \mathcal{F}_{X'}$ (where $X$ and $X'$ are sets of variables), $\varepsilon[X/u]$ is yet another rational expression. Thus, Equation (2.1) is not a definition; on the contrary, this eq. corresponds to a proposition about the (Real) evaluation of expressions resulting from generalized evaluations. For that reason, we stated Equation (2.1) as a lemma in PVS and proved its correctness:

```
1  % Auxiliary definition - image of u under X
2  vars(X: finite_set[variable], u: generalized_evaluation(X)):
       finite_set[variable] =
3       {x: variable | EXISTS (y: (X)): member(x, vars(u(y)))}
4
5  % Equation (2.1)
```

```
evaluations_composition: LEMMA
    FORALL (e: rat_expr,
            u1: generalized_evaluation(vars(e)),
            u2: evaluation(vars(vars(e), u1))):
        (FORALL (x: (vars(e))): real_evaluation?(u1(x))(restrict(u2)))
          IMPLIES
        (eval(gen_eval(e, u1), restrict(u2))
           =
         eval(e, LAMBDA (x: (vars(e))): num(eval(u1(x), restrict(u2)))))
```

The if-then-else operator for expressions is defined as an almost literal transcription of the original definition, thanks to the support for operator overloading in PVS:

```
% Definition 30 - ITE operator for expressions
ITE(x: variable, e1, e2: rat_expr): rat_expr = x*e1 + (1 - x)*e2
```

Using generalized expression evaluation and the if-then-else operator for expressions, we were able to specify the variability encoding function $\gamma$ by translating its original definition:

```
% Definition 31 - Variability encoding function for expressions
gamma(cm: compositional_reliability_model)
     (e: {e: rat_expr | EXISTS (p: (cm`P)): e = alpha_v(p)})
        : RECURSIVE rat_expr =
    LET e_i = LAMBDA (x: (vars(e))): alpha_v(idt_inv(x))
    IN
      gen_eval(e, LAMBDA (x: (vars(e))): ITE(x,
                                             gamma(cm)(e_i(x)),
                                             const(1)))
MEASURE e BY expr_dep(cm)
```

In the above definition, `idt_inv` is the inverse of the identifying function ($idt^{-1}$) and `expr_dep` is the well-founded relation over expressions that is induced by the dependency relation over corresponding PMCs:

```
expr_dep(cm: compositional_reliability_model)
        (e1, e2: rat_expr): boolean =
    EXISTS (p1, p2: (cm`P)): (alpha_v(p1) = e1
                              AND alpha_v(p2) = e2
                              AND cm`dep(p1, p2))
```

### Feature-family-product-based Strategy

The remaining strategy—feature-family-product-based analysis—does not depend on variability encoding of PMCs. Thus, we were able to define it and prove its soundness by means of the already specified elements. In this case, the PVS status for the soundness

theorem is `proved - incomplete`, since it depends on the soundness theorem for the feature-product-based analysis, which is itself marked as incomplete (see Section 4.2.4).

### PMC Encoding

We defined the if-then-else (ITE) operator for PMCs using the same incremental strategy that we used for PMC composition (Section 4.2.4): an ITE operator for disjoint transition matrices, followed by an ITE for disjoint PMCs, and then ITE for arbitrary PMCs (making use of PMC renaming). We also defined the variability encoding function for PMCs (Definition 28) and stated all of the related lemmas and theorems.

However, the main soundness theorem (Theorem 8 – r-equivalence of variability encoding and derivation by composition) required us to state and prove a number of lemmas regarding the preservation of state reachability and of reachability probabilities under ITE composition. For instance, we needed the following lemma, stating that probabilities in the "consequent" part of the ITE (Line 10) remain the same after composition (Line 8):

```
1  ITE_disj_preserves_probabilities1: LEMMA
2      FORALL (x: variable,
3              p1: (compositional_PMC),
4              p2: {p: (compositional_PMC) | disjoint?(p`S, p1`S)},
5              p: {p_: (compositional_PMC) | p_ = ite_disj(x, p1, p2)},
6              s1: (p1`S),
7              u: (well_defined_evaluation(p))):
8          (prob_set(eval(p, u), s1, p`T)
9             =
10          prob_set(eval(p1, restrict(u)), s1, p1`T))
```

Proving such additional lemmas was initially not possible, since we had left `prob_set` uninterpreted (Section 4.2.1). This level of detail, which was abstracted during most of the mechanization effort, needed to be specified at this point. Hence, we also needed to define paths and path probabilities, as follows.

```
1  % We represent paths in DTMCs as lists of states.
2  % This way we can leverage induction to prove lemmas about paths.
3  path?(d: DTMC)(p: list[state]): INDUCTIVE boolean =
4      CASES p OF
5        null: FALSE,                            % The empty list is not a
           path...
6        cons(s, tail): d`S(s) AND
7                       IF null?(tail)
8                         THEN TRUE                % ... but the unary list is!
9                         ELSE path?(d)(tail) AND successor?(d,
                              s)(car(tail))
10                      ENDIF
```

```
11      ENDCASES
12
13  % A path such that the only state in T is the last one
14  path_to_reach?(d: DTMC, first: (d`S), T:
        non_empty_finite_set[(d`S)])(p: list[state]): boolean =
15      EXISTS (t: (T)):
16          (path?(d, first, t)(p)  % the path begins in first and ends in t ∈ T
17           AND
18           FORALL (i: below(length(p)-1)): NOT T(nth(p, i)))
19
20  % The probability of a path (state sequence)
21  prob_path(d: DTMC, p: (path?(d))): RECURSIVE probability =
22      CASES p OF
23        null: 0,
24        cons(s, tail): IF null?(tail)
25                          THEN 1      % [7, Def.  10.10]
26                          ELSE trans(trans(d`P)(s))(car(tail)) *
                                prob_path(d, tail)
27                       ENDIF
28      ENDCASES
29  MEASURE length(p)
30
31  % Probability of reaching a set of states
32  prob_set(d: DTMC,
33           s: (d`S),
34           T: non_empty_finite_set[(d`S)]): probability =
35      sigma(paths_to_reach(d, s, T),
36            LAMBDA (p: list[state]): IF path?(d)(p) THEN prob_path(d, p)
                ELSE 0 ENDIF)
```

In this specification, we defined paths as PVS lists, which means that we only model finite path fragments. This definition is stricter than the one given by Baier and Katoen [7], but the probabilities of finite paths are enough to compute reachability probabilities (finite paths are cylinder sets that abstract away a family of infinite ones). Also, the function `sigma` in Line 35 is the PVS specification of sums over infinite sets (i.e., the $\Sigma$ operator). This function is needed since the set of paths between two given states may be infinite (e.g., if there are any cycles).

The mechanized proof of Theorem 8 also required a specification of Lemma 13, as in the handcrafted version. This lemma had been postponed, since its manual proof relied on the state elimination step of the algorithm by Hahn et al. [41] (Definition 4). Moreover, to state Lemma 13 in PVS, we needed to extend the definition of `gen_eval` to parametric transition matrices and to PMCs.

Similar to PMC composition (Section 4.2.4), state elimination was defined incrementally, to ease the proving of facts about the resulting PMCs. The fundamental definition regards the changes in parametric transition matrices after state elimination, which directly corresponds to Definition 4 (State elimination step).

```
eliminate_state_from_matrix(m: parametric_transition_matrix,
                            s: (dom(m))): parametric_transition_matrix =
    (# dom   := remove(s, dom(m)),
       trans :=
           LAMBDA (s1: (remove(s, dom(m)))):
             (# dom   := remove(s, dom(m)),
                trans :=
                    LAMBDA (s2: (remove(s, dom(m)))):
                      IF (trans(trans(m)(s1))(s) /= const(0)
                          AND trans(trans(m)(s))(s2) /= const(0))
                        THEN (trans(trans(m)(s1))(s2)
                             + (trans(trans(m)(s1))(s)
                                * (trans(trans(m)(s))(s2)
                                * (const(1) /
                                   (const(1)-trans(trans(m)(s))(s))))))
                        ELSE trans(trans(m)(s1))(s2)
                      ENDIF
             #)
    #)
```

Building on the above definition, we were able to define state elimination on PMCs, and then the progressive elimination of a finite set of states. These definitions were leveraged to state and prove lemmas regarding the preservation of properties after state elimination, which were needed to prove Lemma 13.

However, such lemmas about preservation of properties required a notion of correctness of the elimination step. To cope with that, we declared an axiom asserting that state elimination preserves the overall reachability probabilities:

```
elimination_step_is_sound: AXIOM
    FORALL (p: PMC,
            T: {T_: finite_set[(p`S)] | subset?(p`T, T_)},
            s: {s_: (p`S) | s_ /= p`s0
                            AND NOT T(s_)
                            AND not_sink(p, T, s_)},
            u: (well_defined_evaluation(p))):
        eval(alpha_v(p), u)
          =
        eval(alpha_v(eliminate_state(p, T, s)), restrict(u))
```

This fact is a loop invariant of Algorithm 1 that is both proved and used within the proof of Lemma 1 in the work by Hahn et al. [41].

### Preservation of Variables and Revisited Decisions

During the specification of lemmas about state elimination, we needed a result stating that variables in the PMC are preserved:

```
% First attempt...
eliminate_state_from_matrix_preserves_vars: LEMMA
    FORALL (m: parametric_transition_matrix,
            s: (dom(m))):
        vars(eliminate_state_from_matrix(m, s)) = vars(m)
```

However, when proving this lemma, we reached a case in which variables are not preserved: the elimination of *sink* states (i.e., states whose only outgoing transition is a loop, also called *absorbing* states). An example of such case is depicted in Figure 4.2. If the only occurrence of variable $x$ is the transition $s \xrightarrow{t} s_{sink}$, then eliminating $s_{sink}$ will also eliminate that variable, since there is no successor to $s_{sink}$ other than itself.

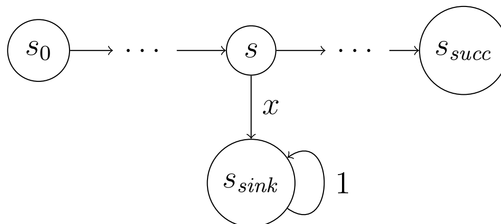



Figure 4.2: Elimination of state $s_{sink}$ “loses” variable $x$

Hahn et al. [41] deal with this by “cropping”—i.e., by making the target states absorbing and removing states (and corresponding edges) that are not reachable from $s_0$ or that cannot reach the target. The problem with this approach is that it may induce a *sub-stochastic* PMC, allowing outgoing transitions of any state to not sum up to 1. Although the algorithm by Hahn et al. [41] is able to handle this, our PVS specification used a base definition of DTMCs that requires them to be stochastic at all times.

One possible solution was to change the definitions of PMC and DTMC to allow sub-stochastic and super-stochastic chains, as in the reference work [41] (although such non-stochastic chains are not covered by Baier and Katoen [7]). However, changing those definitions at this point would require extensive refactoring and would not be cost-effective. Thus, we employed an alternative approach: to require (as a premise) that any input compositional PMC is such that no state (other than the *success* and *error* states) is a sink.

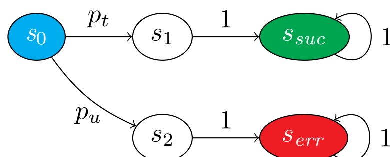
```
% A state that is reachable from the source and from which the
% target states are also reachable.
% This definition relies on loop-free paths, but it should be safe to do so.
% (Because every loop can be eliminated by taking the loop-hole transition directly.)
not_sink(m: parametric_transition_matrix,
          s: (dom(m)),
          source: (remove(s, dom(m))),
          T: {S: finite_set[(dom(m))] | NOT S(s)}): boolean =
     (EXISTS (p1: (non_repeating?)): path?(m, source, s)(p1))
     AND
     (EXISTS (p2: (non_repeating?)): path?(m, s, T)(p2))

not_sink(p: PMC,
          T: {T_: finite_set[(p`S)] | subset?(p`T, T_)},
          s: {s_: (p`S) | s_ /= p`s0 AND NOT T(s_)}): boolean =
     not_sink(p`P, s, p`s0, T)

% Definition of state elimination with the new premise
eliminate_state_from_matrix_preserves_vars: LEMMA
    FORALL (m: parametric_transition_matrix,
              s: (dom(m)),
              source: (remove(s, dom(m))),
              T: {S: finite_set[(dom(m))] | NOT S(s)}):
         not_sink(m, s, source, T)    % This new precondition was required!
           IMPLIES vars(eliminate_state_from_matrix(m, s)) = vars(m)
```

We consider that this premise about non-sink states is consistent. In fact, the hand-crafted version of our theory already posed a requirement for a PMC $\mathcal{P}$ to be considered compositional: that the success and error states are $\mathcal{P}$'s only bottom strongly connected components. Since any sink state is, by definition, a bottom strongly connected component, the existence of a sink state other than the success and error states would lead to a contradiction.

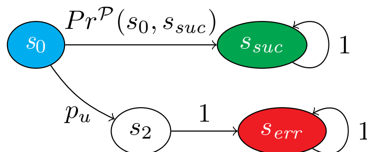
Moreover, we use a notion of user-oriented software reliability whereby we only model (and analyze) a success or a failure (cf. Section 2.2). Since we use transitions to represent module failure and transfer of control between system modules [18], any group of states that is unable to eventually reach either the success or error states represents a deadlock, which can be itself considered a failure. In this case, one can add transitions from the deadlocked states to the error one, making the resulting model free of sink states other than its interface. Also, previous work employed model-driven approaches to generate reliability models from UML behavioral diagrams [36, 54]. The resulting models satisfy the assumption that the only sink states are the success and error states, which raises our confidence that this requirement is realistic (i.e., it does not constrain the generality of

our analysis strategies).

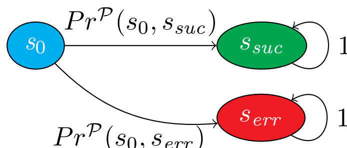
Nonetheless, the handcrafted version of our theory implicitly relied on the state elimination algorithm being executed for **all** states in a given PMC. In reality, however, the algorithm by Hahn et al. [41] (Algorithm 1) assumes that all states are reachable from the starting state and are able to reach at least one state in the target set T. Since our definition of parametric reachability ($\hat{\alpha}$, Definition 21) fixed the target set as the singleton whose only element is $s_{suc}$, the reachability restriction meant that we could not eliminate states that can reach the error state but not the success state.




(a) Compositional PMC $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$




(b) $\mathcal{P}$ after strict state elimination ($T = \{s_{suc}\}$)




(c) $\mathcal{P}$ after the corrected state elimination ($T = \{s_{suc}, s_{err}\}$)

Figure 4.3: Intuition for lemmas regarding state elimination

To illustrate the problem, consider Figure 4.3. The PMC depicted in Figure 4.3a is a compositional PMC $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$ according to Definition 10. Using Algorithm 1 and our definition of $\hat{\alpha}$ strictly, we can only eliminate state $s_1$, since this state reaches $T = \{s_{suc}\}$. However, we cannot eliminate state $s_2$, since it only reaches $s_{err}$. Thus, we can state that the rational expression in the transition $s_0 \rightarrow s_{suc}$ is $Pr^{\mathcal{P}}(s_0, s_{suc})$, but we can make no further assumptions about the remaining transitions (Figure 4.3b).

By the above reasoning, the manual proof of Lemma 13 (r-equivalence of total composition and evaluation) can be deemed wrong. Still, we can overcome this liability by considering the set of target states to comprise both $s_{suc}$ **and** $s_{err}$ for the execution of Algorithm 1. Then, for a compositional PMC $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$, after state elimination is complete, there will be only two transitions left, as assumed in Lemma 13:

$s_0 \xrightarrow{Pr^{\mathcal{P}}(s_0,s_{suc})} s_{suc}$ and $s_0 \xrightarrow{Pr^{\mathcal{P}}(s_0,s_{err})} s_{err}$ (e.g., Figure 4.3c). This approach is considered safe, since we have required that $s_{suc}$ and $s_{err}$ be the only bottom strongly connected components, which means that all states in $\mathcal{P}$ are able to reach either one of them.

To sum up, we found out that the *denotational* semantics of $\hat{\alpha}$ (`alpha_v` in PVS) remains the same—i.e., parametric reachability probability for the success state. On the other hand, the *operational* semantics must change to be that of applying Algorithm 1 using $T = \{s_{suc}, s_{err}\}$ as the input target set, and then selecting the transition $s_0 \xrightarrow{Pr^{\mathcal{P}}(s_0,s_{suc})} s_{suc}$ to comply with the denotational semantics. This way, we can guarantee that the computation of $\hat{\alpha}$ performs the elimination of all states, aside from the interface states $s_0$, $s_{suc}$, and $s_{err}$.

## 4.3 Mechanization Effort

At the present moment, the mechanization of our theory of product-line reliability analysis strategies amounts to 1,176 theorems/lemmas, 1,171 of which are proved. Of these, 782 are TCCs—i.e., proof obligations that are automatically generated by the PVS type checker; the remaining 394 theorems/lemmas were specified by us. A total of 372 of the generated TCCs (approximately half of them) were automatically discharged by the PVS prover. The remaining TCCs, as well as the human-specified theorems/lemmas, were proved interactively using PVS's proof strategies [79]. These totals are summarized in Figure 4.4.

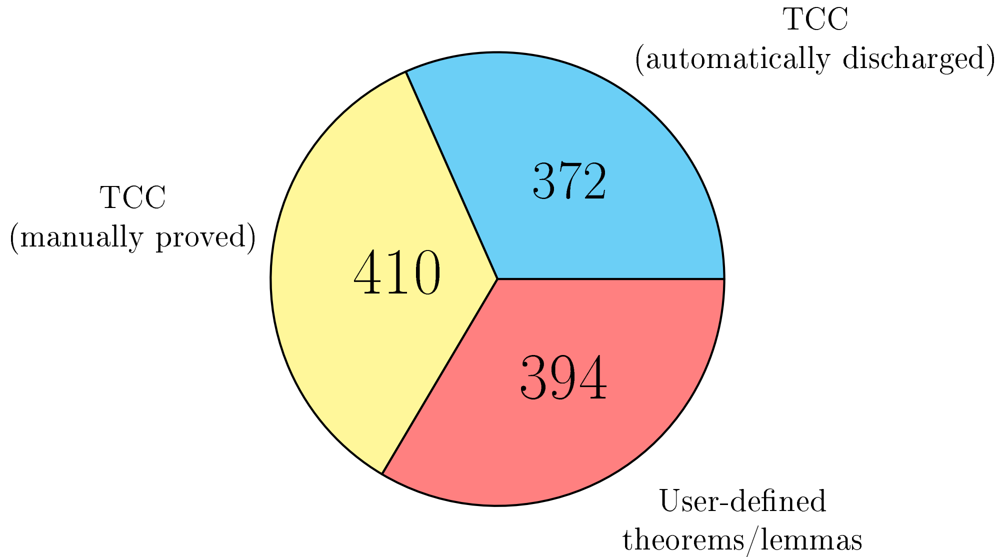



Figure 4.4: Proportion of specification and proof effort

Thus, we can say that PVS handled approximately 2/3 of the theorem *specification* effort, and automated about 1/3 of the *proof* effort. In particular, termination lemmas in

the original theory were automatically generated by PVS as TCCs. Furthermore, the type checker and the proof assistant allowed us to identify gaps in the manual specification. Such gaps represented details that were originally overlooked, either because concrete example models mislead our manual specification or because we considered some constraints implicitly.

However, the total amount of machine-verified lemmas is larger than the total of handcrafted lemmas by a factor of 50. If TCCs are excluded, we still have 15 times more statements proved in PVS than in the original theory. In the following sections, we discuss the distribution (Section 4.3.1) and origin (Section 4.3.2) of such new facts. For the sake of that discussion, we use *lemmas* to also refer to theorems and TCCs. Then, we examine how such lemmas were proved using PVS prover commands (Section 4.3.3) and the benefits of machine-verified proofs through the evolution of our specification (Section 4.3.4).

### 4.3.1 Distribution of Lemmas

Our machine-verified theory is composed of 40 PVS theories distributed throughout 33 files. Every lemma or definition that has a counterpart in the original theory is annotated accordingly, by means of PVS comments. This mapping from handcrafted artifacts to PVS is presented in Table C.2. In Appendix C, we also provide a brief description of the theories (Table C.1) and a map of their relationship (Figure C.1). In this section, we limit the discussion to PVS theories that have more than 20 lemmas.

Figure 4.5 presents such theories, in descending order of size. This figure shows the comparative size of each theory in terms of the overall percentage of lemmas (blue bars). We also present the percentage of lemmas from the handcrafted version that are specified and proved in each theory (red bars). Absolute quantities are annotated besides each bar.

The first thing to notice is that the `PMC_composition` theory alone accounts for approximately **17%** of the mechanized lemmas, despite not presenting any of the original ones. Indeed, this theory presents only two concepts from the handcrafted version: the definitions of total and partial PMC composition (Definitions 12 and 34). The size of this particular theory can be explained by two factors: the need for incremental specification and the proliferation of type-correctness conditions (144 out of the 200 lemmas in `PMC_composition` are TCCs).

The manual definition of total PMC composition (Definition 34) abstracts away important details about the simultaneous composition over many different slots. In Section 4.2.4, we discuss this fact and describe our solution to the problem: incremental specification of composition over a single slot (for transition matrices, for disjoint PMCs, and then for arbitrary PMCs), followed by composition over a set of different slots for a

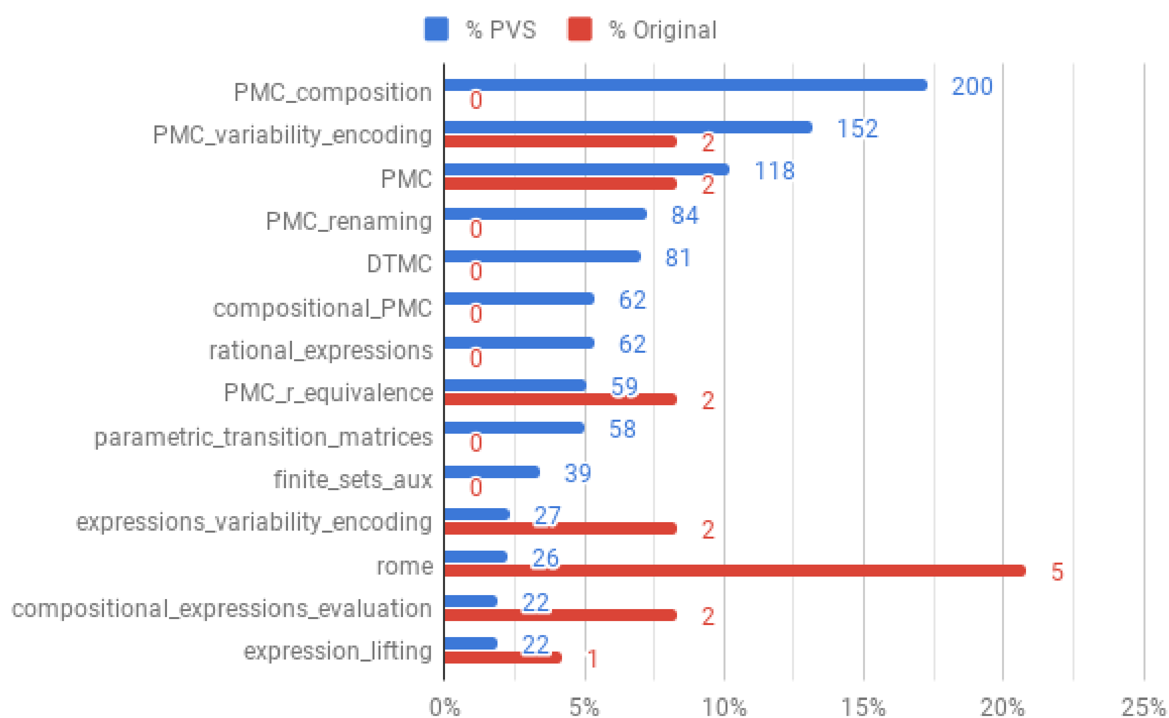


Figure 4.5: Comparison of PVS theories regarding percentage of mechanized lemmas *versus* percentage of the original lemmas

single variable, ending with the composition over sets of slots for different variables. Furthermore, each of these definitions is accompanied by lemmas that state the properties of resulting structures. To ease the proof of these preservation lemmas, we also employed predicate subtyping, which by itself causes proof obligations; nonetheless, most TCCs are generated because we are required to prove that states still belong to the domain of transition matrices (and rows within them) and that the set of variables in the resulting parametric matrices corresponds to the one of the resulting PMC (cf. Section 4.4).

Similar issues occur with theory `PMC_variability_encoding`, which has the second largest set of lemmas. This theory contains the specification of the if-then-else operator for PMCs, which is similar to PMC composition in that both manipulate transition matrices. Accordingly, out of the 152 lemmas in that theory, 117 are TCCs. Moreover, an incremental strategy of specification and proof was also employed in this case.

Theory `PMC`, which ranks third in number of lemmas, formalizes both basic PMC concepts and state elimination. Since the elimination of states manipulates transition matrices, we also used incremental specification and ended up with 88 TCCs among the overall 118 lemmas.

Besides `PMC_composition`, other mechanized theories have no correspondence to *lemmas* in the manual version. Some of these theories are the formalization of *definitions*,

such as PMC_composition itself and PMC_renaming (Definition 33).

Other theories, like parametric_transition_matrices, rational_expressions, DTMC, and ADD, represent the mechanization of third-party concepts. Such theories correspond neither to lemmas nor to definitions of the original theory, but state a number of lemmas and definitions that require explicit formalization in PVS. We believe that these theories may be reused in the formal specification of domains related to our own—using either similar models or analyzing other probabilistic properties.

Last, we have theories that only represent results about foundational concepts such as lists, finite sets, and well-founded relations. These theories are completely agnostic to our domain of interest, so they are the ones with larger reuse potential.

It is worth noting that the rome theory covers more than **20%** of the original theory, but accounts for approximately **2%** of the mechanized lemmas. This is consistent with the fact that rome only contains the definitions of analysis strategies and the corresponding soundness lemmas, which are the key results presented in Chapter 3. Thus, this theory mainly leverages lemmas that are proved elsewhere.

### 4.3.2 Origin of Lemmas

In Section 4.3.1 we briefly discuss one source of lemmas that only exist in the mechanized theory: proof obligations that arise from the manipulation of transition matrices in PMCs. The use of dependent typing in record types accounts for some of such obligations (cf. Section 4.4), but, in general, TCCs are generated as machine-checked conditions for the consistency of the specification. For instance, every specification of recursive function yields a corresponding termination TCC.

Other than automatically generated proof obligations, we have stated 394 lemmas. Among these lemmas, 24 are the lemmas, theorems, and corollaries present in Chapter 3. The others were created to support the mechanized proof of key results and to fill gaps in the handcrafted proofs.

As an example, the original theory assumed that the probabilistic reachability in the feature disabler PMC (Definition 16) is 1. Since this PMC is small (only 3 states and 4 transitions) and its initial state directly reaches the success state, the reader can safely assume that statement to be true. Nonetheless, we had to specify and prove this as two lemmas in PVS, the latter requiring 298 proof commands:

```
1 feature_disabler_is_const: LEMMA
2     FORALL (p: (feature_disabler_PMC?)): const_PMC?(p)
3
4 feature_disabler_reliability_is_1: LEMMA
5     FORALL (p: (feature_disabler_PMC?)): alpha(const_to_DTMC(p)) = 1
```

The previous listing highlights another source of additional lemmas. In the handcrafted proofs, we treated constant rational expressions as Real numbers. However, they have different types and, for that reason, we had to specify conversions in PVS and prove that PMCs can be converted to DTMCs whenever all transitions are constant. For instance:

```
constant_compositional_PMC_const_PMC: LEMMA
    FORALL (p: (compositional_PMC)):
        empty?(vars(p)) IMPLIES const_PMC?(p)
```

Besides the existing gaps in the manually proved theory, we also declared lemmas to support the proof of other results. Some of those lemmas regard the properties of manipulated structures, such as composed PMCs and transition matrices after state elimination (cf. Section 4.3.1). Others refer to more fundamental facts, such as the emptiness of the intersection of a set and its complement ($S \cap \bar{S} = \emptyset$), specified by the following PVS lemma:

```
disjoint_complement: LEMMA
    FORALL (S: set[T]): disjoint?(S, complement(S))
```

To the best of our knowledge, the above fact (and others that we mechanized in theory `finite_sets_aux`) is not available in the built-in PVS libraries (*prelude*) nor in the NASA PVS library.

### 4.3.3 Proof Automation

Our PVS-assisted proofs followed the style of manual proofs as much as possible. However, handcrafted mathematical reasoning is prone to a greater level of abstraction than allowed by the rigor of proof assistants. For that reason, a single step in the original proof usually required a number of proof commands to be fulfilled in PVS.

Lemma 3 (Commutativity of PMC and expression evaluations), for instance, is stated in PVS as follows:

```
eval_commutativity: LEMMA
    FORALL (p: PMC, u: (well_defined_evaluation(p))):
        alpha(eval(p, u)) = eval(alpha_v(p), u)
```

and its mathematical proof consists of applying one lemma and two definitions:

$$\begin{aligned}
\alpha(\mathcal{P}[X/u]) &= \alpha(\mathcal{P}_u) && \text{(syntax change)} \\
&= Pr^{\mathcal{P}_u}(s_0, T) && \text{(Definition 20)} \\
&= \hat{\alpha}(\mathcal{P})[X/u] && \text{(Lemma 1 and Definition 21)}
\end{aligned}$$

The interactive proof in PVS performs almost the same proof steps:

```
1  eval_commutativity :
2
3    |-------
4  {1}   FORALL (p: PMC, u: (well_defined_evaluation(p))):
5          alpha(eval(p, u)) = num(eval(alpha_v(p), u))
6
7  Rule? (skeep :preds? t)      % introduction of Skolem constants
8  Skolemizing and keeping names of the universal formula in (+ -),
9  this simplifies to:
10 eval_commutativity :
11
12 {-1}  well_defined_evaluation(p)(u)      % type constraint preserved by :preds?
13   |-------
14 {1}   alpha(eval(p, u)) = num(eval(alpha_v(p), u))
15
16 Rule? (expand "alpha")       % Definition 20
17 Expanding the definition of alpha,
18 this simplifies to:
19 eval_commutativity :
20
21 [-1]  well_defined_evaluation(p)(u)       % this is a precondition of Lemma 1
22   |-------
23 {1}   prob_set(eval(p, u), eval(p, u)`s0, eval(p, u)`T) =
24         num(eval(alpha_v(p), u))
25
26 Rule? (rewrite "parametric_reachability_soundness" :dir RL)  %Lemma 1
27 Found matching substitution:
28 u:  (well_defined_evaluation(p)) gets u,
29 p:  PMC gets p,
30 Rewriting using parametric_reachability_soundness, matching in *,
31 this simplifies to:
32 eval_commutativity :
33
34 [-1]  well_defined_evaluation(p)(u)
35   |-------
36 {1}   prob_set(eval(p, u), eval(p, u)`s0, eval(p, u)`T) =
37         prob_set(eval(p, u), p`s0, p`T)
```

At this point we are required to perform syntactic manipulations to achieve the desired equality, slightly departing from the manual proof:

```
1  Rule? (expand "eval" 1 (2 3))
2  Expanding the definition of eval,
3  this simplifies to:
4  eval_commutativity :
5
```

```
[-1]  well_defined_evaluation(p)(u)
  |-------
{1}   TRUE

which is trivially true.
Q.E.D.
```

In summary, the PVS proof uses the following commands, which resemble the manual proof:

```
(skeep :preds? t)                                        % Skolem constants
(expand "alpha")                                         % Definition 20
(rewrite "parametric_reachability_soundness" :dir RL)    % Lemma 1
(expand "eval" 1 (2 3))                                  % expand 2nd and 3rd
```

That is not the case for the proof of Theorem 7. The manual proof of this theorem consists of two steps, as follows:

$$\begin{aligned} [\![\hat{\sigma}\big(\mathit{lift}(\hat{\alpha}(\mathcal{P})), \varphi\big)]\!]_c &= [\![\hat{\alpha}(\mathcal{P})]\!]_c^w && \text{(Theorem 6)} \\ &= \alpha([\![\mathcal{P}]\!]_c^{w'}) && \text{(Theorem 5)} \end{aligned}$$

The mechanized proof follows the same structure of the manual proof (modulo syntactic manipulation) up to a certain point:

```
(skeep :preds? t)
(expand "feature_family_based_analysis")
(use "soundness_of_expression_evaluation_using_phi") % Theorem 6
(beta)
(expand restrict)
(rewrite -)                                       % rewrites with Theorem 6
(use "feature_product_soundness")                 % Theorem 5
(expand "feature_product_based_analysis")
(expand restrict)
(expand restrict)
(rewrite - :dir RL)                               % rewrites with Theorem 5
```

At this point, we are left with the following sequent:

```
feature_family_soundness :

[-1]  sink_free_reliability_model?(scm)
[-2]  is_finite(c)
[-3]  [||](fm)(c)
  |-------
{1}   num(sigma(alpha_v(pmc(root(scm))),
                LAMBDA (c: ([|fm|])):
                  LAMBDA (s: (vars(alpha_v(pmc(root(scm)))))):
```

```
10                       the_compositional_evaluation_factory(scm)(c)(s),
11                   c))
12          =
13          num(sigma(alpha_v(pmc(root(scm))),
14                    LAMBDA (c: ([|fm|])):
15                      LAMBDA (s: (vars(root(scm)))):
16                        the_compositional_evaluation_factory(scm)(c)(s),
17                    c))
18
19  Rule?
```

Note that the two sides of the equality in the consequent are practically the same, except for the types of the lambda abstractions (Lines 9 and 15). We have lemmas that help us prove that those types are actually the same, but this requires us to apply functional extensionality. That causes the proof to spawn 4 branches, one of which subdivides 3 times.

Overall, the machine-verified proofs in our theory fall in the same case as the proof of Theorem 7. That is, the main proof branch resembles the manual proof, with auxiliary commands to perform syntactic manipulation and adjustment of types. Nonetheless, there are usually parallel branches for proofs of functional extensionality (to conform type parameters) and preconditions of instantiated lemmas.

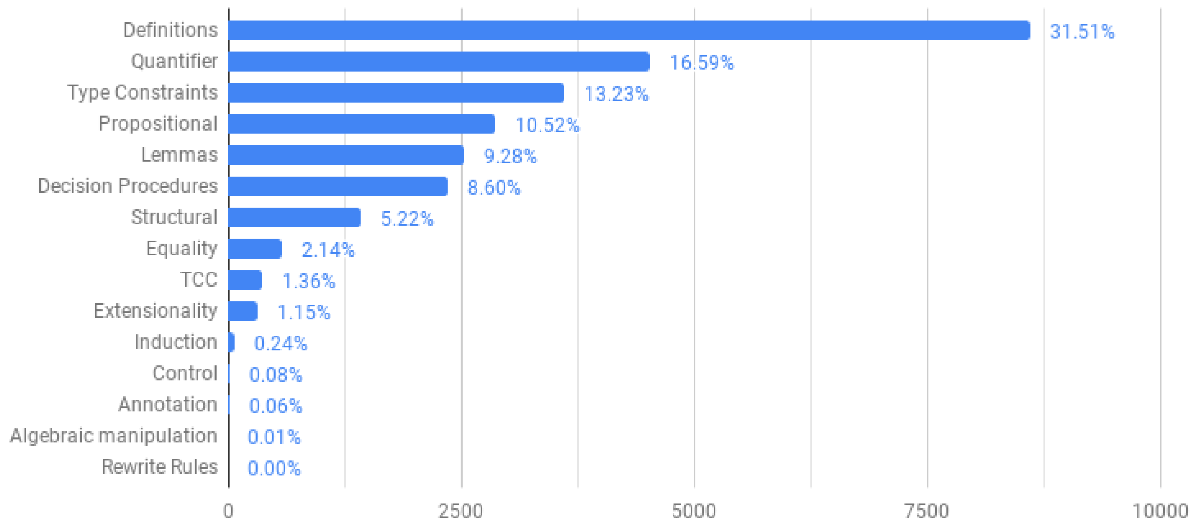



Figure 4.6: Usage of prover commands per category

Figure 4.6 shows the total number of PVS proof commands used in our mechanized theory, grouped by the categories in which they fall (according to the PVS Prover Guide [79]).[12] According to that table, approximately 13% of all proof commands are re-

[12] The complete usage count per category and per command is shown in Tables C.3 and C.4.

lated to introducing type constraints (`typepred`) and using them to rewrite other formulas in the sequent (`rewrite`). Such rewriting is performed by direct use of equational type constraints introduced by `typepred` or by first expanding the definitions of type predicates already present in the sequent.

These pre-existing predicates are brought into the sequent by flags that are passed to the commands that introduce Skolem constants for universally quantified variables (`skeep` and `skolem`), which fall into the category of quantifier rules (along with `inst` and similar rules for elimination of existential quantifiers). The expansion of (new or pre-existing) predicates is handled by the commands `expand` and `expand*`, which comprise the category of definition expansion. This category accounts for about 31% of the commands and also covers the expansion of function definitions.

Together, quantifier rules, expansion of definitions, and manipulation of type constraints account for more than half of the proof commands employed in our PVS theory (around 61%). Furthermore, we also made extensive use of direct formula manipulation using rules for equality and propositional logic (approximately 13% when combined). In other words, roughly 74% of the commands we use in our mechanized proofs can be directly related to proof steps that are (implicitly or explicitly) used in handcrafted demonstrations.

The use of lemmas is another category of prover commands that correspond to a manual proof technique. These commands consist of either direct rewriting (using the `rewrite` rule) or by first instantiating a lemma (by means of `use` or `lemma` followed by `inst`) and then applying `rewrite` or `replace`. Overall, around 9% of the commands correspond to using lemmas as proof steps, which is approximately 1/5 of the amount of explicit expansions of functions and type constraints.

Proof assistants are known to require more detailed specification and proofs than purely mathematical theories—the latter are assessed according to varying degrees of rigor, depending on the reader. Nonetheless, such tools for machine verification also provide appropriate abstractions to cope with some of the inherent complexity, like modules of different granularities (as lemmas and theories), parameterized theories, and automated decision procedures. Thus, the lemma-to-expansion ratio of 1/5 suggests that our mechanized proofs may not be leveraging PVS's strengths as much as it could. This may also be the reason why our proofs often had to be refactored during the evolution of the theory: these proofs relied mostly on direct manipulation of predicates and definitions, instead of using lemmas to abstract such level of detail.

Still, about 9% of the commands used in our theory are automated decision procedures. The most frequently used, `assert`, performs arithmetic and Boolean simplification. This command was employed in case analysis, to discard branches that would lead to con-

tradictions. Besides `assert`, we used `grind` to automatically complete proof branches by means of repeated rewrites and simplifications (leveraging `assert`, among other commands). However, `grind` may leave the sequent in a state where all predicates and functions are expanded to the most basic definitions, rendering the proof more difficult than it originally was. This command may also lead to non-terminating rewrites, which requires a reset of PVS.

We also consider a separate category of automated decision procedures, which we call *TCC commands*. These commands are the ones that PVS assigns as candidate proofs of the type-correctness conditions automatically generated after type checking a theory. Similar to `grind`, these commands either prove a TCC in a single step or enter an infinite chain of rewriting. Nonetheless, such TCC decision procedures were able to automatically discharge almost half of the TCCs in our PVS theories.

Hence, although PVS has powerful built-in decision procedures, they do not apply in every case. Nevertheless, this level of automation reduces the burden of mechanized proofs.

### 4.3.4 Theory Evolution

One of the key goals of a mechanized specification is to have a computer check that theorem proofs are sound. Nonetheless, this goal is not static in time; once a theorem is machine-verified, it can be re-checked every time the corresponding specification is changed. Thus, mechanized proofs also aid along the evolution of the theory.

Throughout the mechanization effort, we followed a design principle to specify only the concepts that were needed, and to do it as the need arises (Section 4.1). This decision meant that our specification suffered frequent refactoring. Still, at every refactoring we were able to use the PVS commands `M-x prove-theory` and `M-x prove-importchain` to re-run the existing mechanized proofs in batch mode. If a proof passed, that meant it was not affected by the change; otherwise, we manually inspected the proof to search for prover rules that failed to apply.

Overall, proof failure after a refactoring meant that either (a) the refactoring was incorrect, or (b) the refactoring changed the number or the ordering of type predicates that were used as sequent formulas in the proof. In the former case, we did not find a pattern to help on identifying a solution; that is object of future work. On the other hand, the latter case could always be identified by backtracking incomplete proof branches.

Such backtracking can be performed by opening a buffer with the proof commands (`M-x show-proof`) and running that proof along with a graphic representation of the proof tree (`M-x x-prove`). For each incomplete proof branch, we start with the leaf node and identify the proof command that led to that sequent. Then, we search for that same

command in the proof commands buffer, to establish a correspondence. Last, we follow the chain of commands in the proof buffer and the path towards the root node in the proof tree in a pairwise fashion, until a mismatch is found.

This command that exists in the recorded proof steps but is not present in the current proof tree has failed to apply. In our experience, this kind of failure is always caused by a proof command that directly references a sequent formula, such as `(rewrite -3)`; if a refactoring changes the number or the ordering of a type constraint that appears as a sequent formula, replaying that command will try to rewrite with a formula that is different from the intended one (despite being indeed at position `[-3]`).

Hence, the solution is to examine the failed command (which was found through backtracking) and discover which formula was needed for this command to succeed. Then, we continue to follow the path towards the root node, seeking for the sequent in which that formula was removed. As mentioned above, this usually occurs at the closest group of `rewrite`s.

## 4.4 Lessons Learned

This section presents an experience report that we think may be useful to researchers willing to perform mechanized specification. In what follows, we discuss some of the design decisions in hindsight, reflecting about the perceived benefits, shortcomings, and alternatives.

**Corrections to the original theory:** Overall, the mechanization effort helped us raise the level of confidence that our theory is sound. Using PVS, we were able to identify and correct gaps in the original (manual) specification. Although those gaps represented liabilities for some of the handcrafted proofs, we were able to refine our theory and correct the detected flaws. Furthermore, solving these issues increased the precision of our specification, since some concepts (e.g., expression composition and simultaneous PMC composition) were informally defined.

It is interesting to note, however, that errors in the original theory appeared in two flavors during mechanization. The first one is that the handcrafted specification may be incomplete, in which case the mechanized version fails to type-check. This sort of mistake is easier to detect, since the PVS type-checker points it out. A concrete example that appeared in our mechanization effort was the need for defining a new generalized expression evaluation function (function `gen_eval`, Section 4.2.5).

On the other hand, there are errors which prevent some proofs from being completed. In such situations, it may not be clear whether a new lemma could enable the proof to

be carried out or if there is indeed a specification flaw. For that reason, these errors are harder to recognize, and their solutions may require a non-localized refactoring.

**Type-correctness conditions (TCC):** PVS type checker was able to automatically generate proof obligations for facts that are important to the soundness of the theory. For instance, function termination may be easily overlooked in a manual specification, but, since PVS requires all functions to be terminating, the type checker forces us to prove this fact. Indeed, our manual specification did not provide a proof of termination for one of the recursive functions.

Also, recursive definitions that make use of predicate subtypes give rise to TCCs requiring that such predicates be preserved at each recursion step. For complex recursions, like the ones that specify PMC composition (cf. Section 4.2.4), stating properties as predicates of the return types may be easier than proving the same facts as independent lemmas.

**Partial functions with predicate subtypes:** We were also able to leverage the type system to mechanically enforce some pre-conditions. For instance, most of our definitions require that evaluation functions be well-defined for the given PMCs. This concept is so pervasive that we dropped explicit mentions to well-definedness whenever we talk about evaluations (Section 2.2.1). Nevertheless, we still need to prove that custom evaluations are indeed well-defined. Hence, we defined a predicate to model well-definedness and used it in all function parameters related to evaluation of PMCs. This way, the type checker generates obligations that prove evaluations to be well-defined. Similar to termination, this also led us to find an evaluation function that was not proved to be well-defined in the original theory.

However, these benefits come at the cost of more complex TCCs. Although PVS decision procedures were able to discharge a considerable proportion of the proof obligations it generated, some of these obligations would not be necessary but for predicate subtypes. Moreover, some TCCs that were not automatically discharged would become simpler (and thus dischargeable by PVS) without predicate subtypes.

One approach to eliminate TCCs would be to specify appropriate judgements. However, the extensive use of dependent typing made it difficult to specify useful judgements in some cases. This happens because of a limitation of PVS, whereby constant judgements do not support dependently-typed quantified variables in the type expression (i.e., after the `HAS_TYPE` keyword).

**Sets as types:** Using sets to define predicate subtypes helped us to design readable definitions, as expected. Nonetheless, one needs to proceed with caution when using this

technique. In our case, we used set types to instantiate some third-party libraries, one of which is NASA's `finite_sets_sum` (which we use to prove facts about finite sums). Because of that, we needed to define auxiliary lemmas for seemingly trivial facts, like the following:

```
1 finite_sum_of_subset: LEMMA
2   FORALL (S1: finite_set[T],
3           S2: {S: finite_set[T] | subset?(S1, S)},
4           f: [(S2) -> real]):
5     sum[(S2), real, 0, +](restrict[T, (S1), boolean](S1), f)
6       =
7     sum[(S1), real, 0, +](restrict[T, (S1), boolean](S1), restrict(f))
```

If we strip type parameters, we can see that this lemma is actually stating an identity of the sum $\sum_{s \in S} f(s)$. Thus, this is only needed because we used a predicate subtype to instantiate the theory, instead of using the base type.

Overall, we can summarize that predicate subtypes are useful to define partial functions, but should be avoided when instantiating theory parameters. Another situation where predicate subtyping should be avoided is when defining parameters of predicates; this over-restricts the domain upon which the predicate can be tested, whereas the intended result is usually that all elements of the base type can be tested.

**Proof automation:** Whenever possible, we used PVS facilities that automated part of the hard work (e.g., the `grind` prover rule and the `typecheck-prove` command). Some rules that try to infer an appropriate instantiation of quantifiers—such as `inst?` and `use`—are preferred instead of explicit instantiation; they require less references to names and sequent formulas, thus making the proof more resilient to changes. However, sometimes the instantiation inference does not fail, but neither produces a useful result.

When working with partial functions, in particular, it is often the case that an appropriate instantiation needs a restricted version of some function $f$ in the proof sequent. In such cases, `inst?` usually uses $f$ without restriction, yielding an impossible obligation to prove that the different domains are equal. Thus, even when applying rules with that automation level, one must inspect the results before proceeding to the next command, under risk of reaching a dead end. These rules alleviate the mental burden, but do not eliminate it.

**Specification style:** The design decision to favor the operational style of specification resulted in the need to expand definitions in proofs. However, some of our definitions—especially the ones related to renaming and composition of matrices—involve multiple conditionals (`IF` and `COND`). This led to proofs with many branches and case analyses,

which can be difficult to design and understand. An alternative strategy would be to specify the overall framework using a declarative style, then specify operational definitions and prove that they conform to the declarative ones. With this alternative approach, we expect to have more definitions and lemmas than using the purely operational style. However, we also expect the resulting proofs to be less involved.

**Function overloading:** Throughout the mechanization process, we used function overloading whenever possible, to group conceptually related functions. In our experience, this led to shorter (but still meaningful) names and improved readability. However, TCCs are named using incremental indices for entities with the same name within a given theory. An overloaded function (such as `rename`, in PMC_renaming) may give rise to a handful of TCCs (about 25, in this particular case), and a change in the definition of one overloaded version of this function may cause the addition or suppression of some of them—causing the following TCCs to change names in cascade.

Since PVS uses the name of a theorem to keep track of its proof (stored in a separate file), this causes a mismatch between the existing proofs and the intended TCC statements. This mismatch must be manually fixed, a task which is both tedious and error-prone. Nonetheless, our view is that the improved readability pays off. Then, we recommend that the specifier take a snapshot of the TCC statements and corresponding proofs (using the commands `M-x tcc` and `M-x show-proofs-theory`) before making changes to overloaded functions.

**Lemmas *versus* brute-force proofs:** Proof commands in PVS can give rise to a number of proof branches, depending on the usage and context. For instance, at some point an ongoing proof had branches for which the reasonable proof strategy was to use induction. However, because of all predicates that applied to the induction variable in that concrete case, issuing the command `induct` spawned 17 branches.

In this case, we removed the concrete context by stating the fact we were trying to prove in that branch as a separate auxiliary lemma. This way, we were able to simplify both the original (encompassing) proof and the new (extracted) one. This indicates that lemmas are useful not only as a reuse technique, but also as a tool to tame proof complexity.

**Using lemmas as interface contracts:** To prove lemmas in PVS, it is useful to preserve type constraints when introducing Skolem constants for universally quantified variables.[13] If the type at hand satisfies a conjunction of predicates, as in the return

[13]PVS quantifier rules, like `skolem`, usually accept Boolean arguments for this purpose.

type of `compose_many_variables` (Section 4.2.4), each of theses predicates turns into a formula in the antecedent. This means that changes in the number (or in the order) of predicates may impact proofs that use them.

Thus, when designing the proof to Lemma 13, we performed a top-down proof approach—i.e., we first designed the lemmas that should be true and then stated these lemmas and designed their proof. Specification of new *definitions* was then postponed by means of a number of lemmas stating the properties needed from them.

Using this approach, we were able to proceed with a proof as far as possible, then quit the prover, specify a new property that was needed, and then resume the proof. If we had chosen to specify these properties directly as predicate subtypes of the function of which the lemma was about, we could have needed to change other proofs. That is, we ended up needing less rework during the initial exploration phase.

The lesson here indicates that lemmas may be useful not only as a reuse technique, but also as a tool to perform incremental specification.

**Limitations of uninterpreted elements:** At several points in our mechanized specification we employed uninterpreted constants and functions (Section 4.2). This technique is useful to abstract details, especially for concepts that are not key to the theory being specified (e.g., propositional rules of feature models) or functions for which there may be many possible definitions (e.g., the parametric model checking function `alpha_v`).

When leaving theory elements uninterpreted, however, one should care to also specify its properties. In our experience, specifying properties of uninterpreted elements at a later time may have a large refactoring impact, since new predicates on the type of a Skolem constant may change the numbers of sequent formulas and many useful prover commands (such as `instantiate` and `rewrite`) rely on them.

Hence, we recommend that every property envisioned for the uninterpreted element be specified using type predicates.

**Limitations of record types:** We used PVS `RECORD` types to specify definitions that use tuples in the handcrafted version of our theory. For instance, we followed this approach with Definition 1 (Parametric Markov Chain), whose PVS counterpart is defined as a direct translation (Section 4.2.1).

However, in a PMC $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$, the set $X$ of variables is dependent on the *actual* variables of the parametric transition matrix $\mathbf{P}$. The explicit mention to both $X$ and $\mathbf{P}$ in the handcrafted version is useful to the reader as variables to which proofs and other definitions refer. In the mechanized version, on the other hand, the specification of both $X$ and $\mathbf{P}$ and the encoding of their relationship as subtype predicates gives rise to

a number of recurrent TCCs at every point where the theory manipulates PMCs. Those TCCs that require us to prove that $X$ = `vars(P)` are usually not hard to prove, but neither do they contribute to the overall soundness of the theory.

So, we suggest that definitions that make use of dependent typing between record members be thoroughly examined, to highlight such cases of over-specification.

## 4.5 Limitations and Threats to Validity

The mechanized theory covers all original aspects of the handcrafted version presented in Chapter 3. However, for scoping reasons, we did not provide machine-verified proofs of auxiliary results that could be traced to background literature. We divide such abstracted results in two categories: axioms and unfinished mechanized proofs.

### 4.5.1 Axioms

Facts that could be referenced in a proof without further arguments (e.g., numbered theorems in books and papers), as well as third-party results, were stated as PVS axioms. Since this type of construct is a potential threat to the consistency of mechanized specifications [67], we tried to avoid them as much as possible, favoring the use of definitions and theorem premises.

Still, our PVS specification uses a total of 7 axioms. Section 4.2 discusses 4 of them in context:

- `reachability_probability_property` (DTMC.pvs, line 173), corresponding to Property 1 (Reachability probability for DTMCs). This property is a result presented in the book by Baier and Katoen [7] at page 760.
- `parametric_reachability_soundness` (PMC.pvs, line 54), corresponding to Lemma 1 (Parametric probabilistic reachability soundness). This lemma is a result from the work by Hahn et al. [41] (Lemma 1).
- `elimination_step_is_sound` (PMC.pvs, line 306), corresponding to Definition 4 (State elimination step). This also comes from the work by Hahn et al. [41], being a loop invariant of Algorithm 1 that is both proved and used within the proof of Lemma 1.
- `alpha_v_eval_is_real` (PMC.pvs, line 49), corresponding to an implied definition in the work by Hahn et al. [41].

The remaining 3 axioms are properties of cylinder sets and sigma algebras, and are only needed in our formalization of DTMCs. These axioms can all be traced to Section 10.1.1 (*Reachability Probabilities*) of the book by Baier and Katoen [7], as follows.

`finite_path_fragments_are_countable` **(DTMC.pvs, line 132).** This axiom states that, for any given DTMC, the set of path fragments going from a state $s$ to a non-empty finite set $T$ of states is countable:

```
1  IMPORTING sigma_set@sigma_countable[list[state]]
2
3  finite_path_fragments_are_countable: AXIOM
4      FORALL (d: DTMC,
5              s: (d`S),
6              T: non_empty_finite_set[(d`S)]):
7          is_countable(paths_to_reach(d, s, T))
```

*Rationale.* We define DTMC paths as finite lists of states, so such paths are guaranteed to be finite. Also, the set of finite path fragments in a DTMC is countable [7], and the set `paths_to_reach(d, s, T)` is a subset of such set. Since every subset of a countable set is also countable, the stated axiom is safe. □

`alternative_paths_sum_up_to_1` **(DTMC.pvs, line 144).** This axiom states that the sum of the (possibly infinite) probabilities for all paths starting in a given state is also a probability:

```
1  alternative_paths_sum_up_to_1: AXIOM
2      FORALL (d: DTMC,
3              s: (d`S),
4              T: non_empty_finite_set[(d`S)]):
5      LET the_sum = sigma(paths_to_reach(d, s, T),
6                          LAMBDA (p: list[state]):
7                              IF path?(d)(p) THEN prob_path(d, p) ELSE 0
                                  ENDIF)
8      IN the_sum >= 0 AND the_sum <= 1
```

*Rationale.* This infinite sum corresponds to the one in the definition of the probability of eventually reaching a set $B$ of states in a DTMC [7]. By definition, this probability is a number in the closed interval $[0, 1] \in \mathbb{R}$. □

`prob_path_is_convergent` **(DTMC.pvs, line 138).** This axiom is needed as a type-correctness condition for the previous one, since the PVS theory of infinite sums is only defined for convergent series:

```
prob_path_is_convergent: AXIOM
    FORALL (d: DTMC,
            s: (d`S),
            T: non_empty_finite_set[(d`S)]):
      convergent?(paths_to_reach(d, s, T))
          (LAMBDA (p: list[state]):
             IF path?(d)(p) THEN prob_path(d, p) ELSE 0 ENDIF)
```

*Rationale.* Again, this result comes from the fact that path probabilities are a probability measure over cylinder sets of the given DTMC, so their sum is convergent [7]. □

## 4.5.2 Unfinished Mechanized Proofs

Besides the previously discussed axioms, there are some auxiliary results that are intuitive, but are not proved elsewhere. In these cases, we stated the results as five ordinary lemmas in PVS, including them initially within the scope of mechanization. The corresponding machine-verified proofs, however, are work in progress.

Those unfinished lemmas were (indirectly) leveraged to prove results regarding r-equivalence (Lemma 13 and Theorem 8). Still, in what follows, we provide non-mechanized proofs of their correctness.

### Sink States and PMC Composition

In the handcrafted version of our theory, we leverage the correctness of the algorithm by Hahn et al. [41] (Lemma 1) to prove that PMC composition and evaluation are r-equivalent (Lemma 13). As already discussed in Section 4.2 (Section 4.2.5), this argument was incomplete and needed the notion of reachability to either the success or the error states. Nonetheless, since we require that these particular states be the only bottom strongly connected components, such reachability is guaranteed [7, Theorem 10.27]. However, we still need to prove that this property is preserved under PMC composition:

```
% Auxiliary definition: a predicate stating that a PMC has
% no sink states besides the success and error states.
no_sink_besides_interface?(p: (compositional_PMC)): boolean =
    FORALL (s: {s_: (p`S) | s_ /= p`s0 AND NOT add(p`s_err, p`T)(s_)}):
        not_sink(p, add(p`s_err, p`T), s)

% Auxiliary definition: a composition that only yields PMCs
% that satisfy the above predicate.
no_sink_besides_interface_composition?(p: (compositional_PMC))
                                      (u: composition(p`X)): boolean =
```

```
11      FORALL (x: (p`X)):
12          no_sink_besides_interface?(u(x))
13
14  composed_states_are_not_sink: LEMMA
15      FORALL (p: (no_sink_besides_interface?),
16              u: (no_sink_besides_interface_composition?(p))):
17        FORALL (s: {s_: (compose(p, u)`S) | s_ /= p`s0
18                                            AND NOT add(p`s_err,
                                                compose(p, u)`T)(s_)}):
19          not_sink(compose(p, u), add(p`s_err, p`T), s)
```

Lemma composed_states_are_not_sink (PMC_r_equivalence.pvs, line 24) states that, if no state in the base PMC is a sink, and if no state in any composed PMC is a sink, either, then we know that no state is a sink in the PMC resulting from composition.[14]

*Proof sketch.* Suppose that this statement is false. Then there is at least one state $s$ in the resulting PMC $P$ that cannot reach either $s_{suc}$ or $s_{err}$. But $s$ comes either from the original (base) PMC $P_{base}$ or from one of the composed PMCs, say $P_{comp}$. So, one of the following must be true:

- If $s \in P_{comp}$, then $s$ reaches either the success or the error states in $P_{comp}$. But, after composition, these states are connected to a slot state from $P_{base}$, which *can* reach either $s_{suc}$ or $s_{err}$. Hence, we have a contradiction.

- Otherwise, if $s \in P_{base}$, then it could reach the interface states before composition. Since composition only changes transitions within a single slot (Definition 12), the interruption of reachability must have happened within a composed PMC $P_{comp}$. However, composition does not affect transitions within the composed PMC besides the loops in the success and error states. Thus, path fragments belonging to composed PMCs are preserved, which means that paths originally passing through the slot are not interrupted. This is, again, a contradiction.

Since both possibilities lead to contradictions, the statement is true. □

Similar reasoning applies to ITE_of_compositional_model_preserves_not_sink (PMC_variability_encoding.pvs, line 478), which states that recursive composition using the PMC if-then-else operator (variability encoding) does not introduce sink states:

```
1  ITE_of_compositional_model_preserves_not_sink: LEMMA
2      FORALL (cm: (sink_free_reliability_model?),
3              p: (cm`P)):
4          no_sink_besides_interface_composition?(p)
```

[14] That description excludes interface states, since the success and error states are sinks by definition.

```
5            (LAMBDA (x: (vars(p))): ite(x,
6                                     gamma(cm)(idt_inv[cm`P, cm`I,
                                          cm`idt](x)),
7                                     feature_disabler_PMC))
```

Again, this lemma assumes that all compositional PMCs in the compositional probabilistic model are free of sink states. Since we are interested in models of system reliability, and since our notion of reliability is binary (i.e., the system either fails or succeeds), we consider this assumption to be safe (cf. Section 4.2.5).

### State Elimination

Lemma `eliminate_composed_maps_slots_to_prob_sets` (PMC_r_equivalence.pvs, line 56) relates directly to the induction hypothesis used to prove Lemma 13:

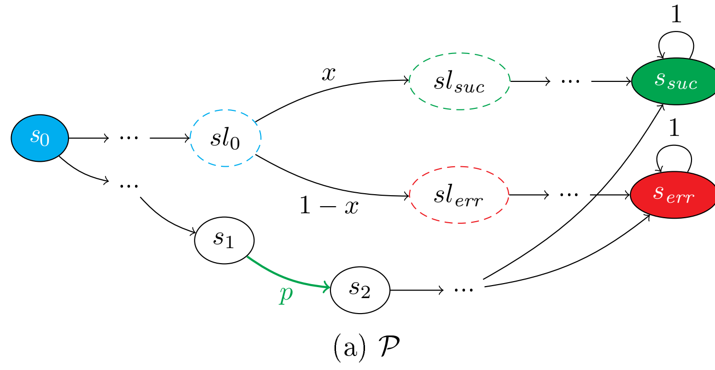
```
1 eliminate_composed_maps_slots_to_prob_sets: LEMMA
2     FORALL (p: (no_sink_besides_interface?),
3             x: (p`X),
4             sl: (slot?(p, x)),
5             u: (no_sink_besides_interface_composition?(p)),
6             u_: (well_defined_evaluation(compose(p, u)))):
7         LET (sl_0, sl_suc, sl_err) = sl IN
8         eval(trans(trans(eliminate_composed(p, u)`P)(sl_0))(sl_suc),
            restrict(u_)) = eval(alpha_v(u(x)), restrict(u_))
9         AND
10        eval(trans(trans(eliminate_composed(p, u)`P)(sl_0))(sl_err),
            restrict(u_)) = eval(const(1) - alpha_v(u(x)), restrict(u_))
```

This lemma establishes that, after applying the state elimination step of Hahn's algorithm to all the states that were added to a PMC by a composition in a single slot, the remaining transitions (between states in said slot) have expressions that are extensionally equal to the reliability expression obtained by using Hahn's algorithm ($\hat{\alpha}$) in the composed PMC. Note that the lemma declaration employs the pattern of specifying extensional equality of rational expressions by comparing the results of applying the `eval` function to each of them.

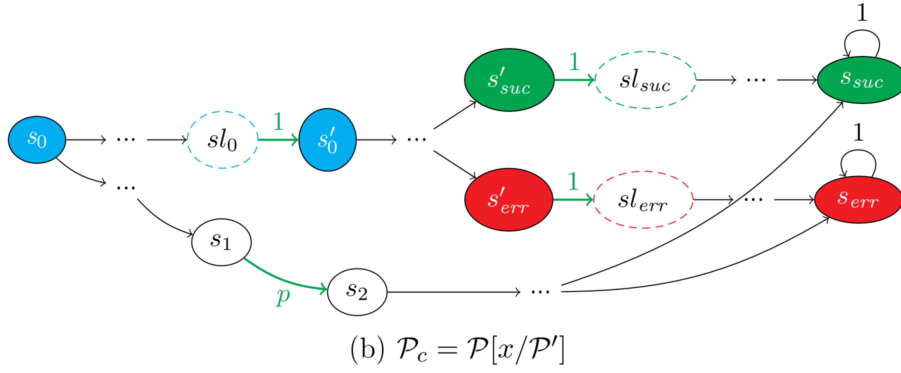
The non-mechanized proof of this statement is already present in the original theory (Lemma 13) and is illustrated by Figure A.1. In Figure 4.7, we can see the evolution from a base PMC $\mathcal{P}$ (Figure 4.7a), going through composition of a PMC $\mathcal{P}'$ (Figure 4.7b) until the elimination of all composed states (Figure 4.7d).

Lemma `eliminate_composed_preserves_non_slots` (PMC_r_equivalence.pvs, line 67), on the other hand, states that transitions between states that are not members of the same slot are preserved after eliminating all states resulting from composition:

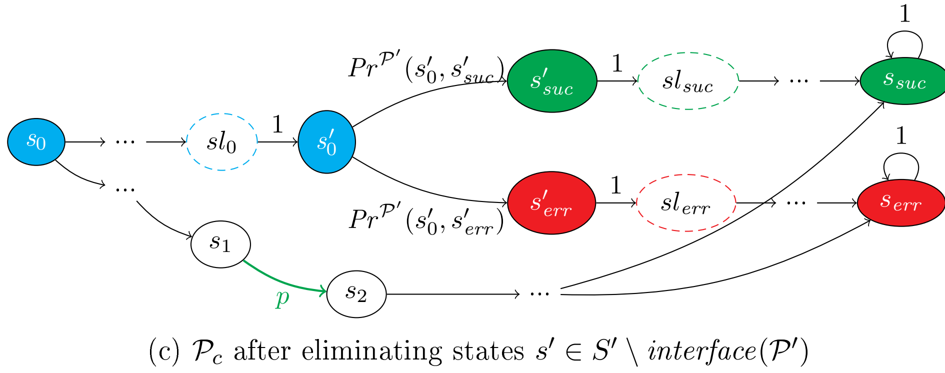
```
1 eliminate_composed_maps_preserves_non_slots: LEMMA
```



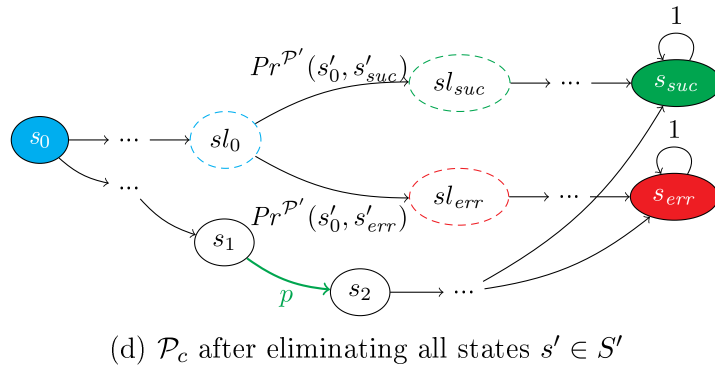

(a) $\mathcal{P}$




(b) $\mathcal{P}_c = \mathcal{P}[x/\mathcal{P}']$




(c) $\mathcal{P}_c$ after eliminating states $s' \in S' \setminus \mathit{interface}(\mathcal{P}')$




(d) $\mathcal{P}_c$ after eliminating all states $s' \in S'$

Figure 4.7: Intuition for lemmas regarding state elimination

```
2       FORALL (p: (no_sink_besides_interface?),
3               s1, s2: (p`S),
4               u: (no_sink_besides_interface_composition?(p)),
5               u_: (well_defined_evaluation(compose(p, u)))):
```

```
6          (NOT EXISTS (sl: (slot?(p))): (s1=sl`1 AND (s2=sl`2 OR
              s2=sl`3)))
7            IMPLIES
8          eval(trans(trans(eliminate_composed(p, u)`P)(s1))(s2),
              restrict(u_)) = eval(trans(trans(p`P)(s1))(s2), restrict(u_))
```

*Proof sketch.* By Definition 12, PMC composition preserves transitions between composed states and only creates transitions between the composed interface and the corresponding states in the target slot. Composition itself does not affect any transition of the base PMC that is not within a slot (as the transition $s_1 \xrightarrow{p} s_2$ in Figure 4.7a). □

Lemma eliminate_state_from_matrix_preserves_wf_evaluations (PMC.pvs, line 175) is the last one regarding state elimination. This lemma states that eliminating a single state preserves the stochasticity of the resulting PMC under well-defined evaluations of the original one.

```
1 eliminate_state_from_matrix_preserves_wf_evaluations: LEMMA
2     FORALL (m: parametric_transition_matrix,
3             s: (dom(m)),
4             source: (remove(s, dom(m))),
5             T: {S: finite_set[(dom(m))] | NOT S(s)},
6             u: (wf_evaluation(m))):
7          not_sink(m, s, source, T)
8            IMPLIES wf_evaluation(eliminate_state_from_matrix(m, s))(u)
```

*Proof sketch.* Let $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$ be a PMC and let $\mathcal{P}' = (S \setminus \{s\}, s_0, X, \mathbf{P}', T)$ be the PMC resulting from the elimination of a state $s \in S \setminus T \setminus \{s_0\}$. We want to prove that, for all $p \in Pre_{\mathcal{P}}(s)$ and for all evaluation $u$ that is well-defined for $\mathcal{P}$, the following holds:

$$\sum_{t \in S \setminus \{s\}} \mathbf{P}'[X/u](p, t) = 1$$

Let $p \in S$ be such that $p \in Pre_{\mathcal{P}}(s)$ (i.e., $p$ is a predecessor of $s$ in $\mathcal{P}$). Any successor of $p$ in $\mathcal{P}'$ must be a state that was previously a successor of both $p$ and $s$, or a successor of just one of them.

$$\begin{aligned} Succ_{\mathcal{P}'}(p) = & \underbrace{(Succ_{\mathcal{P}}(p) \setminus Succ_{\mathcal{P}}(s) \setminus \{s\})}_{S_p \text{ (successors of } p \text{ alone)}} \\ & \cup \underbrace{(Succ_{\mathcal{P}}(p) \cap Succ_{\mathcal{P}}(s) \setminus \{s\})}_{S_{p,s} \text{ (successors of both } p \text{ and } s)} \\ & \cup \underbrace{(Succ_{\mathcal{P}}(s) \setminus Succ_{\mathcal{P}}(p) \setminus \{s\})}_{S_s \text{ (successors of } s \text{ alone)}} \end{aligned}$$

Hence,

$$\begin{aligned}\sum_{t \in S \setminus \{s\}} \mathbf{P}'(p,t) &= \sum_{t \in Succ_{\mathcal{P}'}(p)} \mathbf{P}'(p,t) \\ &= \sum_{t \in S_p} \mathbf{P}'(p,t) + \sum_{t \in S_{p,s}} \mathbf{P}'(p,t) + \sum_{t \in S_s} \mathbf{P}'(p,t)\end{aligned}$$

But, for all $t \in S_p$, $\mathbf{P}'(p,t) = \mathbf{P}(p,t)$, since these transitions are not affected by state elimination. Thus, $\sum_{t \in S_p} \mathbf{P}'(p,t) = \sum_{t \in S_p} \mathbf{P}(p,t)$. Moreover, by definition of the state elimination step (Definition 4), we have that

$$\begin{aligned}\sum_{t \in S_{p,s}} \mathbf{P}'(p,t) = \quad & \mathbf{P}(p,s) \cdot \frac{1}{1 - \mathbf{P}(s,s)} \cdot \mathbf{P}(s,t_0) + \mathbf{P}(p,t_0) \\ + \quad & \mathbf{P}(p,s) \cdot \frac{1}{1 - \mathbf{P}(s,s)} \cdot \mathbf{P}(s,t_1) + \mathbf{P}(p,t_1) \\ + \quad & \ldots \\ = \quad & \mathbf{P}(p,s) \cdot \frac{1}{1 - \mathbf{P}(s,s)} \cdot \sum_{t \in S_{p,s}} \mathbf{P}(s,t) + \sum_{t \in S_{p,s}} \mathbf{P}(p,t)\end{aligned}$$

and

$$\begin{aligned}\sum_{t \in S_s} \mathbf{P}'(p,t) = \quad & \mathbf{P}(p,s) \cdot \frac{1}{1 - \mathbf{P}(s,s)} \cdot \mathbf{P}(s,t_0) + \qquad \cancel{\mathbf{P}(p,t_0)} \\ + \quad & \mathbf{P}(p,s) \cdot \frac{1}{1 - \mathbf{P}(s,s)} \cdot \mathbf{P}(s,t_1) + \qquad \underbrace{\cancel{\mathbf{P}(p,t_1)}}_{\textit{these states are not successors of } p} \\ + \quad & \ldots \\ = \quad & \mathbf{P}(p,s) \cdot \frac{1}{1 - \mathbf{P}(s,s)} \cdot \sum_{t \in S_s} \mathbf{P}(s,t)\end{aligned}$$

Since $\mathcal{P}$ is stochastic under evaluation $u$, it holds that $Succ_{\mathcal{P}}(s) = (Succ_{\mathcal{P}}(s) \cap Succ_{\mathcal{P}}(p)) \cup (Succ_{\mathcal{P}}(s) \setminus Succ_{\mathcal{P}}(p))$ and $\sum_{t \in Succ_{\mathcal{P}}(s)} \mathbf{P}(s,t) = \sum_{t \in S} \mathbf{P}(s,t) = 1$ (omitting the evaluation syntax for brevity). Hence, we have that

$$\begin{aligned}\sum_{t \in S_s} \mathbf{P}(s,t) + \sum_{t \in S_{p,s}} \mathbf{P}(s,t) + \mathbf{P}(s,s) = 1 \\ \sum_{t \in S_s} \mathbf{P}(s,t) = 1 - \sum_{t \in S_{p,s}} \mathbf{P}(s,t) - \mathbf{P}(s,s)\end{aligned}$$

Thus,

$$\sum_{t \in S_s} \mathbf{P}'(p,t) = \frac{\mathbf{P}(p,s)}{1-\mathbf{P}(s,s)} \cdot \left(1 - \sum_{t \in S_{p,s}} \mathbf{P}(s,t) - \mathbf{P}(s,s)\right)$$

and we have that

$$\begin{aligned}
\sum_{t \in S \setminus \{s\}} \mathbf{P}'(p,t) &= \sum_{t \in S_p} \mathbf{P}(p,t) + \sum_{t \in S_{p,s}} \mathbf{P}'(p,t) + \sum_{t \in S_s} \mathbf{P}'(p,t) \\
&= \sum_{t \in S_p} \mathbf{P}(p,t) + \cancel{\frac{\mathbf{P}(p,s)}{1-\mathbf{P}(s,s)} \cdot \sum_{t \in S_{p,s}} \mathbf{P}(s,t)} + \sum_{t \in S_{p,s}} \mathbf{P}(p,t) \\
&\quad + \frac{\mathbf{P}(p,s)}{1-\mathbf{P}(s,s)} \cdot \left(1 - \cancel{\sum_{t \in S_{p,s}} \mathbf{P}(s,t)} - \mathbf{P}(s,s)\right) \\
&= \sum_{t \in S_p} \mathbf{P}(p,t) + \sum_{t \in S_{p,s}} \mathbf{P}(p,t) + \frac{\mathbf{P}(p,s)}{\cancel{1-\mathbf{P}(s,s)}} \cdot \cancel{(1-\mathbf{P}(s,s))} \\
&= \sum_{t \in S_p} \mathbf{P}(p,t) + \sum_{t \in S_{p,s}} \mathbf{P}(p,t) + \mathbf{P}(p,s) \\
&= \sum_{t \in Succ_{\mathcal{P}}(p)} \mathbf{P}(p,t) \\
&= 1
\end{aligned}$$

□

# Chapter 5

# Conclusions

In this work, we formally presented seven approaches to reliability analysis of product lines, covering all strategies in the taxonomy by Thüm et al. [85]. In particular, we formally extended previous work [16] with feature-based analysis strategies. To the best of our knowledge, this is the first work to address all three dimensions of product-line analysis (product-based, family-based, and feature-based) in the context of model checking, and also the first to present an instance of feature-family-product-based analysis strategy.

The soundness of our analysis techniques is established by results on the commutativity of their intermediate steps, summarized by the commuting diagram in Figure 3.7. This constitutes formal evidence that, given a product line, each of the presented approaches yields the same results as the others, enabling practitioners to choose among analysis strategies based on their space and time trade-offs. We had our proofs reviewed by fellow researchers outside our group, and the resulting theory is published by a peer-reviewed journal [15].

Moreover, we mechanized this theory in the PVS proof assistant [67]. The mapping between the theory as presented in Chapter 3 and the mechanized specification in Chapter 4 is summarized in Table C.2, in Appendix C. By now, the mechanized specification (Chapter 4) consists of 1,176 theorems/lemmas, of which 5 auxiliary lemmas (needed in the machine-verified version) are not yet machine-verified. All elements in the original theory are specified and proved. To the best of our knowledge, this is the first work to present a mechanized specification of product-line reliability analysis.

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.

In the remainder of this chapter, we discuss implications of the results (Section 5.1) and limitations of our research (Section 5.2). Then, we discuss related work (Section 5.3)

and conclude by proposing future work (Section 5.4), some of which is in progress.

## 5.1 Discussion of Results

The main contribution of this work relies on the proofs of commutativity and the associated theory. This theory may be leveraged as a starting point to formalize the verification of other quality properties that can be expressed using related Markov models, such as Continuous-time Markov Chains (CTMC) and Markov Decision Processes (MDP). Such leverage may be obtained from possible similarities in the handcrafted theory, but also from refactorings of the mechanized version.

Moreover, the commutativity proofs increase the confidence that the techniques whose *performance* were empirically compared [54] are indeed alternatives to one another. As a corollary, this proves that the feature-family-based analysis technique presented by Lanna et al. [54] is correct.[1] Hence, it is safe to assume that, according to current evidence, in the context of user-oriented reliability analysis using DTMC models of product lines, the feature-family-based strategy outperforms the others.

Still, the theory presented here also extends the analysis strategies assessed by Lanna et al. [54], including a novel feature-family-product-based strategy. This motivates the need for further empirical studies that also cover the variant strategies in this work.

The formalization of our theory in PVS allowed us to identify some errors and imprecisions of the handcrafted version. That is, some of the original results (Chapter 3) were partially invalidated by the mechanization effort. Nonetheless, we were able to correct the identified issues and modify the parts of the theory depending on them. Thus, the key results of the theories presented in Chapters 3 and 4 are slightly different from one another.

Furthermore, the mechanized version has significantly more elements—specifically, there are approximately 50 times more lemmas in PVS than there are in the handcrafted theory. Even though roughly 2/3 of those lemmas were proof obligations automatically generated by PVS, they still had to be proved. Approximately half of these proof obligations were automatically discharged, but the absolute amount of interactive proving exceeded by far the quantity of handcrafted proofs. Also, the process of manually specifying and proving the original theory took about 1 year, whereas the machine-assisted version required 2.5 years so far (despite being based on existing knowledge).

Because the mechanization effort was significantly higher than that of creating the original theory, it is possible to see it as overkill. However, that endeavor brings the

[1] That work actually presents a model-driven method to compute the user-oriented reliability given UML models of a system. However, the results presented here do not apply to the automatic extraction of DTMC models from UML, but only to the analysis of the extracted Markov chains.

benefit of providing a foundation on which similar theories can be built. The process of specifying and proving our theory of commuting strategies in PVS also gave rise to auxiliary theories that can be used in more general contexts—for instance, theories about ADDs, rational expressions, and DTMCs, besides additional lemmas on finite sets and lists. Such auxiliary theories are under evaluation for joining the NASA PVS Library.

Additionally, we believe that the experience report presented in Chapter 4 will be helpful for researchers and practitioners working with interactive theorem proving. We documented the design decisions, the main obstacles, and the mistakes we made, so that the community may also benefit from the lessons learned during the mechanization process.

Overall, this work highlights the relationship between analysis steps in the context of model-checking the user-oriented reliability of product lines. Figure 3.7 depicts the patterns that were found during our research, relating annotative and compositional models as well as the operations defined over them. Such view allows the organization and structuring of facts (e.g., commutativity of intermediate analysis steps) in a concise and precise manner, facilitating the communication of ideas and contributing to a more comprehensive understanding of underlying principles used in these strategies.

Although we do not claim that our results are general enough to be immediately applied to other contexts, these results add up to the knowledge base of theoretical work regarding the formal verification of product lines. Moreover, we believe that the commuting diagram in Figure 3.7 may be leveraged as a guide to the formalization of related theories. This conjecture is mentioned at the ending of Chapter 3 and is object of an ongoing work within our research group. Hence, this work indirectly contributes to the ongoing search for a principle and possibly automated way to lift a given specification and analysis technique to product lines [85].

## 5.2 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 created the original (manual) specification by modeling the constructs that exist in the *ReAna* product-line reliability analysis tool—implemented by our research group

to perform empirical studies [54]. Although this tool is implemented in an imperative object-oriented programming language (Java), it employs a functional programming style as much as possible. 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 [5]. This programming discipline does not *guarantee* a correct mapping between software and mathematical assets, but mitigates the risk of mismatching.

To increase the confidence in the consistency and soundness of our formal definitions and proofs, we submitted the specifications for review by fellow researchers and by anonymous reviewers of a scientific journal, resulting in the paper's acceptance [15]. The publication venue was chosen because the members of its editorial board are experienced in model checking and in the use of formal methods in general, and some of them do research on software product lines.

Besides human scrutiny, we further increase the evidence on the soundness of our commutativity theory by means of machine-based verification. All of the key results are specified and machine-verified, which already indicates that the original theory is probably sound. Nonetheless, correspondence to the manual specification still needs to be established. We mitigate that risk by explicitly providing pointers from the mechanized to the original elements of the theory (Appendix C), and by keeping the translation between both as direct as possible (Section 4.1). In the cases where the mechanized definitions become more involved, we discuss the differences and argue about their rationale (Section 4.2).

Also, our mechanized version relies on 7 axioms, and axiomatic specification brings the issue of whether the resulting theory is inconsistent. Indeed, there is evidence that this may be the case even for experienced PVS practitioners [72]. To reduce that threat, we provided a mapping from our axioms to results available in the literature (Section 4.5) and sought to keep them as syntactically similar to their reference definition as possible. Moreover, 5 of the lemmas in our mechanized theory have unfinished proofs, which hinders their validity. This is indeed an open issue, which we mitigate by providing handcrafted proofs and arguments of correctness.

Our approach to mitigate the risk of inconsistent axioms assumes that the literature results on which we rely are indeed correct. These results are based on the book by Baier and Katoen [7] and on the work by Hahn et al. [41]. Both works have influenced many others, based on the number of citations reported by Google Scholar.[2] The latter work, in particular, laid the foundations for the PARAM model checker [40], whose model checking techniques have been also incorporated in the PRISM model checker [52].

[2]As of November 2019, Baier and Katoen [7] had 4956 citations and Hahn et al. [41] was cited 126 times according to `http://scholar.google.com`.

Another potential risk to the validity of our machine-verified results is the use of third-party theories. Such theories have axioms of their own, which could introduce inconsistencies. However, all theories imported within our own are part of NASA PVS library.[3] Those specification and proof artifacts are publicly available and are curated by a research group (NASA Langley Formal Methods Team). As such, and given that this library is used in formal verification of mission-critical software, we assume that the risk of introducing inconsistencies is low. For the same reason, we consider unlikely that any of our mechanized proofs is falsely deemed correct due to an unnoticed bug in PVS itself.

Last, we must discuss to what extent our results can be generalized. By construction, we limited our scope to user-oriented reliability analysis using model checking. Thus, we do not claim that our results can be immediately generalized to other types of analysis. On the contrary, we suggest that specific research might be conducted towards generalizing our theory (cf. Section 5.4).

## 5.3 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 [85]. In particular, several model checking techniques have been successfully lifted to work with product lines [4, 19, 21, 23, 24, 34, 36, 51, 64, 82]. In contrast to existing research, our work presents different analysis techniques, covering all groups identified in the taxonomy by Thüm et al. [85], and relates these techniques to one another. Moreover, we present what is—to the best of our knowledge—the first feature-family-product-based analysis strategy in the literature. Hence, we discuss the closest related work according to different criteria.

**PMC-based analysis of product lines:** Ghezzi and Sharifloo [36] 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 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 $\mathscr{P}$ of compo-

[3] https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/

sitional PMCs, and the decomposition tree induces the dependency relation $\prec$. For that reason, our work provides formal evidence of the soundness of their approach.

Rodrigues et al. [76] 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) [42]. 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 [85] in its entirety. 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 $(S, \nu, FM, \Pi)$ such that, for all $s, s' \in S$ and $c \in [\![FM]\!]$:

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

Rodrigues et al. [77] applied similar composition techniques to DTMC models of probabilistic reachability analysis. That work also exploits the notion of interfaces to derive a model of the whole system from models of its components. Moreover, the authors propose an algorithm to abstract inner (i.e., non-interface) transitions of component models, such that the state-space explosion inherent to model checking techniques is reduced in the final composite model. However, Rodrigues et al. [77] use this technique as a tool to model a single component-based system closely to its intended architecture, whereas our work leverages composition as a variability representation mechanism.

**Feature-based model checking:** Li et al. [56] and Liu et al. [57] 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) [20], 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. [56] can have

an arbitrary number of outgoing states, and Liu et al. [57] 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. [56] and Liu et al. [57] 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.

**Family-based model checking:** Dubslaff et al. [34] created a framework for modeling probabilistic and nondeterministic 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 [19]. 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 is our focus, it only enables family-based and product-based strategies (which the authors call, respectively, *all-in-one* and *one-by-one* [34]), 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 constraints in the behavioral model itself.

Kowal et al. [50] 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). However, similar to Dubslaff et al. [34], they only address family-based analysis of a model derived from the delta modules. Another difference to our work is that 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 [2, 4, 51, 74]. von Rhein et al. [91] 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 definitions of variability encoding for PMCs and expressions. Furthermore, their overall proof strategy resembles the one used throughout our work (i.e., comparison of results for corresponding configurations). However, whereas von Rhein et al. [91] use trace semantics and a weak bisimulation relation to correlate behaviors, we perform this task using structural analysis of the behavioral models. Despite being less general, structural analysis is sufficiently strong for the purpose of proving that reliability is preserved, which is the main focus of our work.

**Formal approaches to variability-aware analysis:** The definition of product-line analysis techniques that are sound by construction has been investigated recently [12, 14, 17, 62], although not specifically in the context of model checking. Midtgaard et al. [62] 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 general framework for derivation of analysis strategies; 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. Moreover, whereas Midtgaard et al. handle only the family-based dimension of analysis, we also address the feature-based dimension. In this sense, our work can also be seen as a preliminary investigation on deriving alternative strategies to perform a given analysis.

Brabrand et al. [14] proposed a technique to automatically lift intraprocedural data-flow analyses to handle variability in product lines. Similar to our work, the authors propose alternative analysis strategies, which are derived by gradually introducing variability awareness in different components of an existing analysis. Brabrand et al. [14] also

present a soundness proof for the proposed strategies, whereby all of them are guaranteed to compute the same result as the base analysis. The presented simultaneous and consecutive analysis strategies are similar to our family-based and family-product-based ones, respectively, even though different properties are analyzed. However, Brabrand et al. [14] do not consider feature-based analyses. Furthermore, our work breaks down analysis strategies in intermediate steps that can be composed in different ways, enabling reuse of proofs.

**Mechanized specification of product lines:** Other researchers have leveraged theorem provers and proof assistants in the context of software product lines (e.g., Borba et al. [13], Delaware et al. [30], Neves et al. [63], Teixeira et al. [81], Thüm et al. [83]). However, most of the existing work investigates the reuse of specification and proofs to assert soundness of different products in a given product line (product lines of theorems). Our work, in contrast, deals with properties of product lines in general.

Borba et al. [13] also specified a PVS theory about properties of product lines—in their case, for reasoning about safe product-line evolution. That work evolved into a product line of theories [81], where products are theories of safe evolution based on concrete product-line languages. Similar to our results, their work present PVS theories about properties of product lines. Nonetheless, Neves et al. [63] and Teixeira et al. [81] specified concepts in the domain of product-line engineering, whereby the targets of their theories are meta-models of product lines. Our work focuses on properties of product-line reliability analysis strategies, instead. Future work may investigate how to map our specification to concrete product-line languages by leveraging the meta-theory by Teixeira et al. [81].

**Comparison of analysis dimensions:** Kolesnikov et al. [49] 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 that our techniques can be implemented in a tool to perform empirical studies. Furthermore, Kolesnikov et al. neither investigate combined strategies nor prove the soundness of the implemented type checkers.

von Rhein et al. [90] 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. [90] with a formally defined relation among analyses and intermediate steps, albeit restricted to reliability analysis.

**Conceptual models and taxonomy:** Thüm et al. [85] 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. [90] 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). Since our soundness proofs for variability encoding and feature composition apply to single features (not necessarily maximal PMCs), our techniques range over the PLA plane of feature grouping and variability encoding dimensions. Furthermore, 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.

Meinicke et al. [60] 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. [85]), and strategy of the tool (product-based, variability-aware, and variability-encoding). Using this taxonomy, an implementation of our techniques would cover all possibilities on the dimensions of implementation technique, strategies for product-line analysis, and strategy of the tool, while the dimension of analysis technique would be fixed to reliability analysis.

## 5.4 Future Work

The research presented here provided a formal specification of strategies for analyzing software product lines. As such, we visualize that there are opportunities for both ex-

tending the analytical reasoning of our theory and performing further empirical studies that leverage the formal results.

**Further handling of threats to validity:** Although we have mechanized all aspects of the original theory, there are still elements in our PVS code that can be further explored. First and foremost, we plan to fully mechanize the unfinished lemmas, despite the mathematical reasoning that we presented to argue about their correctness (Section 4.5). In a related issue, specifying and proving the work by Hahn et al. [41] and the approach to DTMCs in the book by Baier and Katoen [7] could (a) reduce the need for axioms in our theory and (b) be reused as a foundation on which to perform similar research.

**Extensions to the current theory:** In principle, our theory can be extended to cover other aspects within the same domain (i.e., user-oriented reliability modeled using DTMC). For instance, our commuting diagram could be expanded upwards by formalizing the modeling approaches and the relation between the derivation processes for products and for models thereof. Another possibility is to explore the inverse of variability encoding arrows—that is, how to decompose annotative models into compositional ones, to leverage feature-based strategies in the analysis of existing annotative product lines.

**Generalization of the mechanized theory:** The mechanization of our theory in PVS has led to benefits of its own, especially regarding the refinement and correction of the handcrafted version (Chapter 4) and the general purpose byproducts (e.g., PVS theories for ADD, rational expressions, DTMC, and PMC). Nonetheless, we believe that there are long-term benefits to be gained by generalizing the PVS mechanization of commuting analysis steps to other kinds of quality properties and analysis techniques.

For instance, there are other software quality properties that are of practical interest and can be analyzed using probabilistic models [37]. An alternative view of reliability, for instance, is the probability that a system does not fail within a given time interval [43]. This property can be specified using a probabilistic invariance formula, which is different from the probabilistic existence used in our specific case but can still be checked over DTMC models [37]. On the other hand, one could extend our analysis of user-oriented reliability in the face of nondeterminism by checking the same probabilistic existence property but using Markov Decision Processes (MDP) instead of DTMC models.

The above suggestions of generalization either fix the property being checked and let the model of system behavior vary, or fix the model and change the property. However, one could also think about how different models *and* properties can fit in the same product-line commutative analysis framework. Indeed, the analysis dimensions covered by our work belong to a taxonomy that was derived from an extensive survey [85]. Hence, it

makes sense to investigate analysis commutativity in the context of type checking, dataflow analysis, theorem proving, or even other types of model checking (e.g., checking the absence of deadlocks or race conditions and checking properties of Featured Transition Systems [21]).

Accordingly, we intend to explore to what extent our theory can be generalized to other types of product-line models and analyses thereof. To reach this goal, we plan to iteratively refactor our specification, seeking to abstract away details that are directly related to reliability analysis. The resulting specification will be compared with existing related work, to consider whether the proposed generalization can be instantiated for similar models.

**Empirical studies over the mechanized theory:** Throughout the mechanization process, we encountered obstacles and had to progressively refine our specifications. Nonetheless, some of the lessons from this process would require a deep refactoring of the theory. Thus, future work could perform an exploratory study on alternative design decisions and how to simplify the current PVS theory. With this study, we expect to present a systematic assessment of the impact of different specification and proof styles.

Furthermore, it would be beneficial to the PVS community to investigate and classify specification and proof smells as well as refactoring patterns. It is also of interest to explore how to best evolve the specifications and proofs while avoiding rework—particularly because of explicit references to sequent formulas.

**Empirical studies regarding the analysis strategies:** Given that we provide *analytical* evidence that the strategies presented here are equivalent to one another, there is *empirical* evidence indicating that the feature-family-based strategy outperforms the others [54]. However, the existing empirical study does not take our novel feature-family-product-based strategy into account. Further experiments should be performed to assess the additional strategy and to investigate how the characteristics of subject product-line models impact analysis performance.

Moreover, there is an opportunity to evolve the existing implementation of the reliability analysis strategies [54] to reflect the machine-verified theory. This way, there would be increased confidence that the existing product-line reliability analysis tool corresponds to the strategies that were proven sound.

# Bibliography


[1] 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. 15

[2] 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)*, pages 372–375. IEEE Computer Society, 2011. ISBN 978-1-4577-1638-6. doi: 10.1109/ASE.2011.6100075. 42, 70, 146

[3] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer, 2013. ISBN 978-3-642-37520-0. doi: 10.1007/978-3-642-37521-7. 1, 5, 9, 10, 11, 12, 13, 15, 37, 70, 79, 80

[4] 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. IEEE Press, 2013. ISBN 9781467330763. doi: 10.1109/ICSE.2013.6606594. 16, 143, 146

[5] 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, August 1978. ISSN 00010782. doi: 10.1145/359576.359579. 142

[6] 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. 4, 24, 25, 27, 79

[7] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499. 1, 15, 18, 20, 44, 85, 86, 101, 110, 112, 130, 131, 132, 142, 149, 167

[8] Jonatas Ferreira Bastos, Paulo Anselmo da Mota Silveira Neto, Pádraig O'Leary, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. Software product lines adoption in small organizations. *Journal of Systems and Software*, 131:112 – 128, 2017. ISSN 0164-1212. doi: 10.1016/j.jss.2017.05.052. 1

[9] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.23. 14

[10] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 7:1–7:8, United States, January 2013. Association for Computing Machinery. ISBN 978-1-4503-1541-8. 1

[11] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS series. Springer Verlag, 2004. 82

[12] 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, 146

[13] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. A theory of software product line refinement. *Theoretical Computer Science*, 455:2–30, October 2012. ISSN 03043975. doi: 10.1016/j.tcs.2012.01.031. 147

[14] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. In *Transactions on Aspect-Oriented Software Development X*, pages 73–108. Springer, 2013. ISBN 978-3-642-36964-3. doi: 10.1007/978-3-642-36964-3_3. viii, 2, 146, 147

[15] Thiago Castro, André Lanna, Vander Alves, Leopoldo Teixeira, Sven Apel, and Pierre-Yves Schobbens. All roads lead to Rome: Commuting strategies for product-line reliability analysis. *Science of Computer Programming*, 152:116 – 160, 2018. ISSN 0167-6423. doi: 10.1016/j.scico.2017.10.013. viii, 5, 7, 35, 81, 96, 139, 142

[16] Thiago Mael de Castro. *Estratégias Comutativas para Análise de Confiabilidade em Linhas de Produtos de Software*. Dissertation, Universidade de Brasília, 2016. URL http://repositorio.unb.br/handle/10482/22680. 3, 35, 139

[17] Sheng Chen and Martin Erwig. Type-based parametric analysis of program families. *ACM SIGPLAN Notices*, 49(9):39–51, August 2014. ISSN 03621340. doi: 10.1145/2692915.2628155. 146

[18] R. Cheung. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, 6(02):118–125, March 1980. ISSN 1939-3520. doi: 10.1109/TSE.1980.234477. 1, 3, 18, 113

[19] 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. viii, 2, 15, 20, 36, 143, 145

[20] 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. Springer, 1982. ISBN 3-540-11212-X. doi: 10.1007/BFb0025774. 144

[21] 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, 15, 143, 150

[22] 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. ACM Press, 2010. ISBN 9781605587196. doi: 10.1145/1806799.1806850. 1, 36

[23] 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, 15, 143

[24] 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, February 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2013.09.019. 2, 15, 143

[25] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001. 1, 10

[26] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-30977-7. 1, 11, 12

[27] 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. 12

[28] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Proceedings of the Third International Conference on Software Product Lines (SPLC)*, pages 162–164. Springer, July 2004. ISBN 978-3-540-28630-1. doi: 10.1007/978-3-540-28630-1_17. 94

[29] Conrado Daws. Symbolic and parametric model checking of discrete-time Markov chains. In *Proceedings of the First International Conference on Theoretical Aspects of Computing (ICTAC)*, volume 3407 of *Lecture Notes in Computer Science*, pages

280–294. Springer, September 2005. ISBN 978-3-540-25304-4. doi: 10.1007/b107116. 20

[30] Benjamin Delaware, William Cook, and Don Batory. Product lines of theorems. *ACM SIGPLAN Notices*, 46(10):595, October 2011. ISSN 03621340. doi: 10.1145/2076021.2048113. 147

[31] E W Dijkstra. On program families. In *Notes on Structured Programming*. Academic Press, 1971. 10

[32] 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

[33] 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

[34] 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. viii, 2, 15, 143, 145

[35] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology*, 106:1 – 30, 2019. ISSN 0950-5849. doi: 10.1016/j.infsof.2018.08.015. 1

[36] 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, March 2013. ISSN 09505849. doi: 10.1016/j.infsof.2012.07.017. 2, 15, 20, 36, 113, 143

[37] Lars Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 31–40. ACM, 2008. doi: 10.1145/1368088.1368094. 19, 149

[38] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta modeling for software architectures. In *Proceedings of the 7th Dagstuhl workshop on Model-based Development of Embedded Systems (MBEES)*, pages 1–10, 2011. 14

[39] 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 (SPLC)*, page 22. ACM Press, 2013. ISBN 9781450319683. doi: 10.1145/2491627.2491632. 14, 42

[40] 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. Springer, 2010. doi: 10.1007/978-3-642-14295-6_56. 19, 142

[41] 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. xiii, 20, 22, 23, 24, 53, 55, 89, 92, 93, 106, 110, 112, 114, 130, 132, 142, 149, 166, 167

[42] 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. 19, 144

[43] David I. Heimann, Nitin Mittal, and Kishor S. Trivedi. Availability and reliability modeling for computer systems. volume 31 of *Advances in Computers*, pages 175 – 233. Elsevier, 1990. doi: 10.1016/S0065-2458(08)60154-0. 149

[44] 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, April 2016. doi: 10.1016/j.infsof.2015.11.004. 1

[45] Karam Ignaim and João M. Fernandes. An industrial case study for adopting software product lines in automotive industry: an evolution-based approach for software product lines (EVOA-SPL). In *Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC) - Volume B*, pages 183–190, new york, ny, usa, 2019. acm. ISBN 978-1-4503-6668-7. doi: 10.1145/3307630.3342409. 1

[46] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990. 11

[47] 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. ACM Press, 2008. ISBN 9781605580791. doi: 10.1145/1368088.1368131. 37

[48] 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. 14

[49] Sergiy Kolesnikov, Alexander von Rhein, Claus Hunsen, and Sven Apel. A comparison of product-based, feature-based, and family-based type checking. In *Proceedings of the 12th International Conference on Generative Programming (GPCE)*, pages 115–124. ACM, 2013. ISBN 978-1-4503-2373-4. doi: 10.1145/2517208.2517213. vii, 2, 17, 147

[50] 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. Springer, 2014. ISBN 978-3-642-54803-1. doi: 10.1007/978-3-642-54804-8_7. 145

[51] 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, November 2015. doi: 10.1109/ASE.2015.16. 2, 15, 143, 146

[52] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *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. 19, 142

[53] 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

[54] André Lanna, Thiago Castro, Vander Alves, Genaina Rodrigues, Pierre-Yves Schobbens, and Sven Apel. Feature-family-based reliability analysis of software product lines. *Information and Software Technology*, 94:59 – 81, 2018. ISSN 0950-5849. doi: 10.1016/j.infsof.2017.10.001. vii, viii, 1, 2, 3, 5, 17, 18, 36, 84, 113, 140, 142, 150

[55] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Available at http://www.jmlspecs.org, 2006. 16

[56] 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, July 2005. ISSN 0928-8910. doi: 10.1007/s10515-005-2643-9. 144, 145

[57] 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, December 2010. ISSN 0928-8910. doi: 10.1007/s10515-010-0075-7. 144, 145

[58] Idarlan Machado, Rodrigo Bonifácio, Vander Alves, Lucinéia Turnes, and Giselle Machado. Managing variability in business processes: An aspect-oriented approach. In *Proceedings of the 2011 International Workshop on Early Aspects*, EA '11, pages 25–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0645-4. doi: 10.1145/1960502.1960508. 15

[59] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, ICSE '16,

pages 643–654, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884793. vii, 2, 17, 80

[60] 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. ACM Press, September 2014. ISBN 9781450327398. doi: 10.1145/2647908.2655972. 148

[61] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992. ISSN 0018-9162. doi: 10.1109/2.161279. 16

[62] 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, July 2015. ISSN 01676423. doi: 10.1016/j.scico.2015.04.005. viii, 2, 146

[63] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulezsa, and Paulo Borba. Investigating the safe evolution of software product lines. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering - GPCE '11*, volume 47, page 33, New York, New York, USA, October 2011. ACM Press. ISBN 9781450306898. doi: 10.1145/2047862.2047869. 147

[64] 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, September 2012. doi: 10.1109/SBCARS.2012.23. 2, 15, 143

[65] 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)*, pages 19–22. SBC, April 2013. ISBN 978-85-7669-274-4. 36

[66] Object Management Group. Unified Modeling Language. Available at `https://www.omg.org/spec/UML/2.5/PDF`, 2015. Version 2.5. 18

[67] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001. URL `http://pvs.csl.sri.com/doc/pvs-language-reference.pdf`. 5, 7, 15, 16, 27, 28, 29, 84, 130, 139

[68] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001. URL `http://pvs.csl.sri.com/doc/pvs-system-guide.pdf`. 27, 28, 82

[69] 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. ACM, 2013. ISBN 978-1-4503-1968-3. doi: 10.1145/2491627.2491628. 14

[70] Leonardo Pessoa, Paula Fernandes, Thiago Castro, Vander Alves, Genaína N. Rodrigues, and Hervaldo Carvalho. Building reliable and maintainable dynamic software product lines: An investigation in the body sensor network domain. *Information and Software Technology*, 86:54 – 70, 2017. ISSN 0950-5849. doi: 10.1016/j.infsof.2017.02.002. viii, 2

[71] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 0-262-16209-1. 15

[72] Lee Pike. A note on inconsistent axioms in Rushby's "systematic formal verification for fault-tolerant time-triggered algorithms". *IEEE Transactions on Software Engineering*, 32(5):347–348, May 2006. ISSN 00985589. doi: 10.1109/TSE.2006.41. 142

[73] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. ISBN 3540243720. 1, 10

[74] 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. 70, 146

[75] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997. doi: 10.1007/BFb0053389. 14

[76] 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. xiii, 1, 2, 11, 12, 15, 20, 37, 144

[77] Pedro Rodrigues, Emil Lupu, and Jeff Kramer. Compositional reliability analysis for probabilistic component automata. In *Proceedings of the Seventh International Workshop on Modeling in Software Engineering*, MiSE '15, pages 19–24, Piscataway, NJ, USA, 2015. IEEE Press. 144

[78] 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. Springer, 2010. ISBN 3-642-15578-2, 978-3-642-15578-9. 14

[79] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001. URL http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf. 27, 28, 31, 115, 122

[80] 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. 10

[81] 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. ACM Press, July 2015. ISBN 9781450336130. doi: 10.1145/2791060.2791105. 15, 94, 147

[82] Maurice H. ter Beek and Erik P. de Vink. Towards modular verification of software product lines with mCRL2. In *Proceedings of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 368–385. Springer, 2014. ISBN 978-3-662-45234-9. doi: 10.1007/978-3-662-45234-9_26. 2, 15, 143

[83] 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, March 2011. ISBN 978-1-4577-0019-4. doi: 10.1109/ICSTW.2011.48. 16, 147

[84] 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, April 2013. ISSN 0362-1340. doi: 10.1145/2480361.2371404. 42

[85] 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, June 2014. ISSN 03600300. doi: 10.1145/2580950. vii, 2, 3, 4, 5, 15, 16, 17, 51, 53, 54, 55, 63, 64, 77, 139, 141, 143, 144, 148, 149

[86] 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, September 2011. ISBN 978-0-7695-4626-1. doi: 10.1109/SBCARS.2011.13. 14

[87] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007. ISBN 3540714367. 1, 10

[88] 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, August 2001. ISBN 0-7695-1360-3. doi: 10.1109/WICSA.2001.948406. 10

[89] Karina Villela, Adeline Silva, Tassio Vale, and Eduardo Santana de Almeida. A survey on software variability management approaches. In *Proceedings of the 18th International Software Product Line Conference (SPLC) - Volume 1*, pages 147–156,

New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2740-4. doi: 10.1145/2648511.2648527. 1

[90] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. The PLA model: on the combination of product-line analyses. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 14:1–14:8. ACM Press, January 2013. ISBN 9781450315418. doi: 10.1145/2430502.2430522. 147, 148

[91] 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, January 2016. ISSN 23522208. doi: 10.1016/j.jlamp.2015.06.007. viii, 2, 42, 70, 146

[92] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. Variability-aware static analysis at scale: An empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(4):18:1–18:33, November 2018. ISSN 1049-331X. doi: 10.1145/3280986. vii, 1, 2, 17

[93] 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. ACM Press, October 2014. ISBN 9781450332101. doi: 10.1145/2661136.2661143. 58

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

[95] Jonas Åkesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. Migrating the Android Apo-Games into an annotation-based software product line. In *Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC) - Volume A*, pages 103–107, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-7138-4. doi: 10.1145/3336294.3342362. 1

# Acronyms

**ADD** Algebraic Decision Diagram.

**CTL** Computation Tree Logic.

**CTMC** Continuous-Time Markov Chain.

**DTMC** Discrete-Time Markov Chain.

**JML** Java Modeling Language.

**MDP** Markov Decision Process.

**PCTL** Probabilistic Computation Tree Logic.

**PMC** Parametric Markov Chain.

**PVS** Prototype Verification System.

**SPL** Software Product Line.

**TCC** Type-correctness Condition.

**UML** Unified Modeling Language.

# Appendix A

# Additional Proofs

This appendix contains formal definitions and proofs that were omitted from the main body of the paper to avoid digressions.

## A.1 Existence of Minimal and Maximal PMCs

**Lemma 9** (Existence of minimal PMCs)**.** *Given a set $\mathscr{P}$ of compositional PMCs, an identifying function idt, and the corresponding induced well-founded relation $\prec$, there exists, at least, one* minimal *PMC $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$. Furthermore, $X = \emptyset$, that is, minimal PMCs are, in fact, DTMCs with defined interfaces and only two bottom strongly connected components (cf. Definition 10).*

*Proof.* The existence of minimal PMCs follows directly from the fact that the induced relation $\prec$ is well-founded: otherwise, all descending chains would be infinite.

Now suppose $X \neq \emptyset$. Then, it has, at least, one element $x$. Since the set $I$ of identifiers (image of $idt$) is a superset of all $X_i$, $x \in I$. By definition, function $idt$ is bijective, so there must be a compositional PMC $\mathcal{P}' \in \mathscr{P}$ such that $idt(\mathcal{P}') = x$. But $idt(\mathcal{P}') \in X \Rightarrow \mathcal{P}' \prec \mathcal{P}$. Since $\mathcal{P}$ is minimal by hypothesis, this is a contradiction. □

**Lemma 10** (Existence of maximal PMCs)**.** *Given a set $\mathscr{P}$ of compositional PMCs, an identifying function idt, and the corresponding induced well-founded relation $\prec$, there exists, at least, one* maximal *PMC $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$.*

*Proof.* The proof is by contraposition. Nonexistence of such maximal PMC means there are infinite *ascending* chains $\mathcal{P}_1 \prec \mathcal{P}_2 \prec \mathcal{P}_3 \prec \dots$ for $\mathcal{P}_i \in \mathscr{P}$. Since $\mathscr{P}$ is finite, such infinite chain implies the existence of cycles, that is, at least, one $\mathcal{P}_i$ transitively depending on itself. But cycles imply both ascending and descending infinite chains, contradicting the well-foundedness of $\prec$. Hence, there are no infinite ascending chains under $\prec$ and, by contraposition, there is, at least, one maximal PMC. □

## A.2 Termination Lemmas

The following lemma states the termination of the recursive definitions of the composition factory $w'$ (Definition 18).

**Lemma 11** (Derivation by composition terminates)**.** *For a compositional model* $(\mathscr{P}, \prec, I, idt, p, w', FM)$*, for all configurations* $c \in [\![FM]\!]$*, the composition function* $w'(c)$ *terminates.*

*Proof.* Let $idt^{-1} : I \to \mathscr{P}$ be the inverse function of $idt$. To prove $w'(c)$ terminates, we note that the arguments in the recursive calls in the definition of $w'$ (Definition 18) strictly decrease if we use $idt^{-1}$ as a measure function into the well-founded set $\mathscr{P}$.

Without loss of generality, let $x = idt(\mathcal{P})$ for some $\mathcal{P} \in \mathscr{P}$ with variables set $X = \{x_1, \ldots, x_k\}$. The right-hand side of $w'(c)(x)$ evaluates to either $\mathcal{P}_\perp$ (the feature disabler PMC) or $\mathcal{P}[x_1/w'(c)(x_1), \ldots, x_k/w'(c)(x_k)]$. In the first case, it trivially terminates, since $\mathcal{P}_\perp$ have no slots; in the latter, the arguments to each recursive call are the variables $x_i \in X$. By definition, $x_i = idt(\mathcal{P}_i)$ for some $\mathcal{P}_i \in \mathscr{P}$ such that $\mathcal{P}_i \prec \mathcal{P}$. Thus, $idt^{-1}(x_i) \prec idt^{-1}(x)$. Since $\prec$ is well-founded, $w'(c)$ terminates. □

The following lemma states that the recursion in Definition 26 terminates.

**Lemma 12** (Compositional evaluation terminates)**.** *For a compositional model* $(\mathscr{P}, \prec, I, idt, p, w', FM)$*, for all configurations* $c \in [\![FM]\!]$*, the compositional evaluation* $w(c)$ *terminates.*

*Proof.* Let $idt^{-1} : I \to \mathscr{P}$ be the inverse function of $idt$. To prove $w(c)$ terminates, we note that the arguments in recursive calls to $w(c)$ (Definition 26) strictly decrease if we use $idt^{-1}$ as a measure function into the well-founded set $\mathscr{P}$.

Indeed, without loss of generality, let $x = idt(\mathcal{P})$ for some $\mathcal{P} \in \mathscr{P}$ with variables set $X = \{x_1, \ldots, x_k\}$. By definition of $\sigma$, the right-hand side of $w(c)(x)$ evaluates to either 1 or $\hat{\alpha}(\mathcal{P})[x_1/w(c)(x_1), \ldots, x_k/w(c)(x_k)]$. In the first case, it trivially terminates; in the second, the arguments to each recursive call are the variables $x_i \in X$. By definition, $x_i = idt(\mathcal{P}_i)$ for some $\mathcal{P}_i \in \mathscr{P}$ such that $\mathcal{P}_i \prec \mathcal{P}$. Thus, $idt^{-1}(x_i) \prec idt^{-1}(x)$. Since $\prec$ is well-founded, $w(c)$ terminates. □

## A.3 Soundness of Feature-product-based Analysis

We first state formally what we mean by PMC renaming, which is a key concept in PMC composition.

**Definition 33** (Compositional PMC renaming)**.** Given a compositional PMC $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$, the $i$-th renaming of $\mathcal{P}$, $\mathcal{P}^i = (S^i, s_0^i, s_{suc}^i, s_{err}^i, X^i, \mathbf{P}^i, T^i)$, is an isomorphic compositional PMC with renamed states. That is, $\mathcal{P}^i$ is such that:

- $S^i \cap S = \emptyset$.
- $\forall_{i,j \in \mathbb{N}} \cdot i \neq j \implies S^i \cap S^j = \emptyset$.
- There exists a bijective mapping $_^i : S \to S^i$ from each state $s_j \in S$ to a state $s_j^i \in S^i$.
- $X^i = X$.
- $\forall_{s_1, s_2 \in S} \cdot \mathbf{P}^i(s_1^i, s_2^i) = \mathbf{P}(s_1, s_2)$.
- $T^i = \{s^i \mid s \in T\}$.

With the formal definition of PMC renaming, we are able to present a precise definition of a total composition, obtained by composing PMCs over all slots in a given base compositional PMC at once.

**Definition 34** (Total PMC composition)**.** Given a compositional PMC $(S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$ with $k$ variables $x_1, \ldots, x_k$, and a set $\mathscr{P}$ of $k$ compositional PMCs $(S_i, s_{i_0}, s_{i_{suc}}, s_{i_{err}}, X_i, \mathbf{P}_i, T_i)$, $i \in \{1, \ldots, k\}$, let $u' : X \to \mathscr{P}$ be a function that yields a compositional PMC $\mathcal{P} \in \mathscr{P}$ to compose in the corresponding slots for any given variable. Let also $n_i = |slots^{\mathcal{P}}(x_i)|$ for $i \in 1, \ldots, k$, and $\mathcal{P}_i^j = (S_i^j, s_{i_0}^j, s_{i_{suc}}^j, s_{i_{err}}^j, X_i^j, \mathbf{P}_i^j, T_i^j)$ for $j \in 1, \ldots, n_i$ be the $j$-th renaming of $\mathcal{P}_i$ (Definition 33). The total PMC composition $\mathcal{P}[X/u']$, also denoted by $\mathcal{P}[x_1/u'(x_1), \ldots, x_k/u'(x_k)]$, is a compositional PMC $\mathcal{P}' = (S', s_0', s_{suc}', s_{err}', X', \mathbf{P}', T')$ such that:

- $S' = S \uplus \biguplus_{j=1}^{n_1} S_1^j \uplus \cdots \uplus \biguplus_{j=1}^{n_k} S_k^j$, where $\uplus$ denotes the disjoint union operator (all states are disjointly merged);
- $s_0' = s_0$, $s_{suc}' = s_{suc}$, and $s_{err}' = s_{err}$ (the interface of $\mathcal{P}$ is preserved);
- $X' = \bigcup_{i=1}^{k} X_i$ (each occurrence of $x_i$ is replaced by a copy of $\mathcal{P}_i$, whose variables are those of $X_i$);
- $T' = T$ (target states of the base PMC are preserved);
- $\mathbf{P}'$ is such that, for all slots $(s_{x_{i_0}^j}, s_{x_{i_{suc}}^j}, s_{x_{i_{err}}^j})$ of the base PMC $\mathcal{P}$ and interfaces $(s_{i_0}^j, s_{i_{suc}}^j, s_{i_{err}}^j)$ of the renamed PMCs $\mathcal{P}_i^j$ (where $i \in 1, \ldots, k$ and $j \in 1, \ldots, n_i$),

- $\mathbf{P}'(s_{x^j_{i_0}}, s^j_{i_0}) = 1$ (new transition from a slot's initial state to the initial state of the corresponding composed PMC)
- $\mathbf{P}'(s^j_{i_{suc}}, s_{x^j_{i_{suc}}}) = 1$ (new transition from the success state of a composed PMC to the success state of the corresponding slot)
- $\mathbf{P}'(s^j_{i_{err}}, s_{x^j_{i_{err}}}) = 1$ (new transition from the error state of a composed PMC to the error state of the corresponding slot)
- $\mathbf{P}'(s_{x^j_{i_0}}, s_{x^j_{i_{suc}}}) = 0$ (slot's success transition is removed)
- $\mathbf{P}'(s_{x^j_{i_0}}, s_{x^j_{i_{err}}}) = 0$ (slot's error transition is removed)
- $\mathbf{P}'(s^j_{i_{suc}}, s^j_{i_{suc}}) = 0$ (success loops from composed PMCs are removed)
- $\mathbf{P}'(s^j_{i_{err}}, s^j_{i_{err}}) = 0$ (error loops from composed PMCs are removed)
- For all remaining combinations of $s_1, s_2 \in S'$:

$$\mathbf{P}'(s_1, s_2) = \begin{cases} \mathbf{P}(s_1, s_2) & \text{if } s_1, s_2 \in S \setminus \mathit{slotStates}^{\mathcal{P}}(X) \\ \mathbf{P}^j_i(s_1, s_2) & \text{if } s_1, s_2 \in S^j_i \\ 0 & \text{otherwise} \end{cases}$$

The function $u'$ is called a *composition function.*

To establish the soundness of the feature-product-based strategy, we need to compare it to the product-based strategy for compositional models. However, the latter relies on PMC composition, while the former is based on compositional evaluation of expressions. To bridge this gap, we first note that, as far as reliability analysis is concerned, composing a PMC $\mathcal{P}'$ into a slot of another PMC $\mathcal{P}$ is equivalent to evaluating the corresponding variable in $\mathcal{P}$ with the reliability expression of $\mathcal{P}'$ (i.e., $\hat{\alpha}(\mathcal{P}')$).

**Lemma 13** (r-equivalence of total composition and evaluation)**.** *Let $\mathcal{P}, \mathcal{P}_1, \ldots, \mathcal{P}_k$ be compositional parametric Markov chains, and $X = \{x_1, \ldots, x_k\}$ be $\mathcal{P}$'s set of variables. Then,*

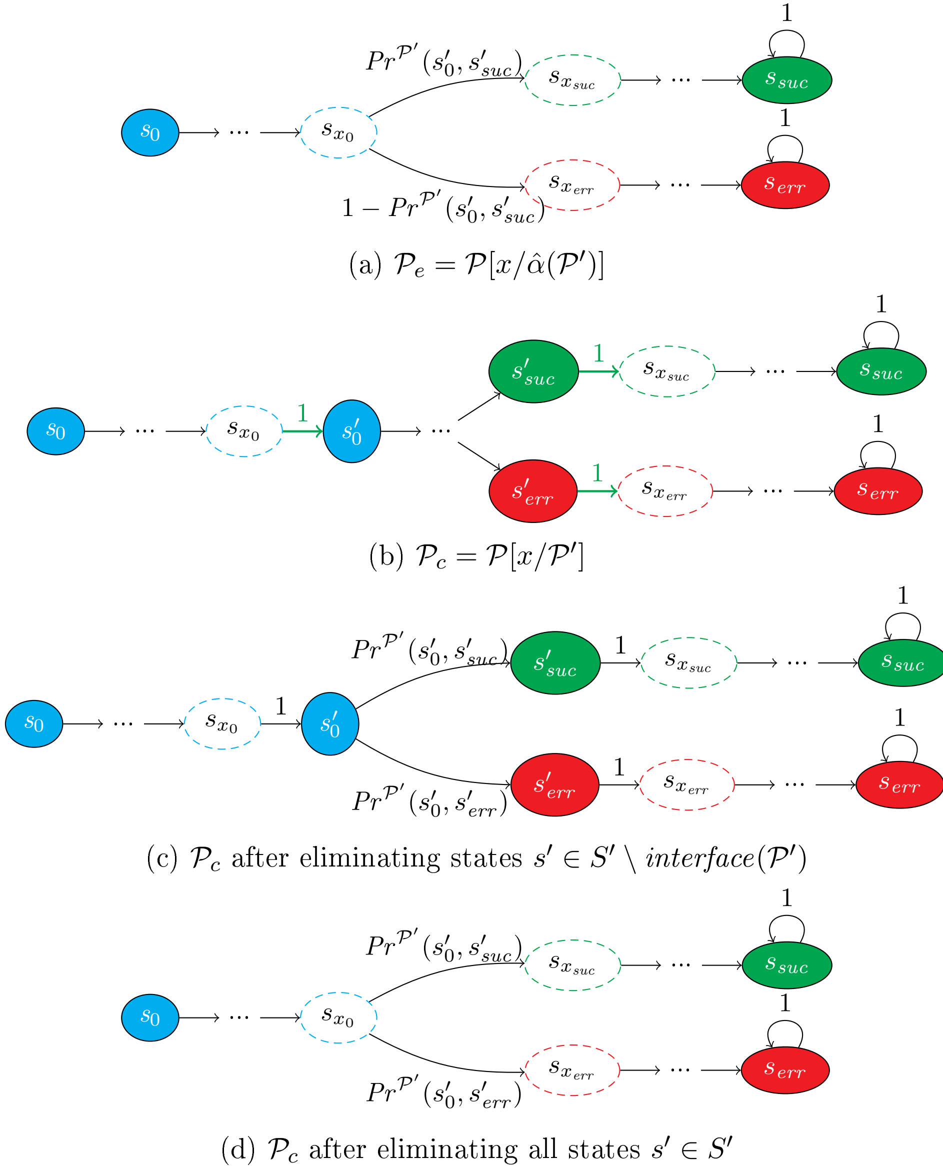
$$\hat{\alpha}(\mathcal{P}[x_1/\mathcal{P}_1, \ldots, x_k/\mathcal{P}_k]) = \hat{\alpha}(\mathcal{P}[x_1/\hat{\alpha}(\mathcal{P}_1), \ldots, x_k/\hat{\alpha}(\mathcal{P}_k)])$$

*where the equals sign denotes* extensional *equality. In other words, the two expressions (i.e., syntactic objects) are not necessarily equal in a syntactical sense, but their corresponding rational functions (i.e., semantic objects) always yield equal values if given equal inputs.*

*Proof.* The main argument for this proof is the case where $\mathcal{P}$ has only one variable, that is, $X = \{x\}$. This way, we start by proving that $\hat{\alpha}(\mathcal{P}[x/\mathcal{P}']) = \hat{\alpha}(\mathcal{P}[x/\hat{\alpha}(\mathcal{P}')])$ for a given

compositional PMC $\mathcal{P}' = (S', s'_0, s'_{suc}, s'_{err}, X', \mathbf{P}', T')$. Then, we extend this to the general case where $\mathcal{P}$ has an arbitrary number of variables.

A generic illustration of $\mathcal{P}$ and $\mathcal{P}'$ is given by Figures 3.4a and 3.4b, respectively. Let $\mathcal{P}_e = \mathcal{P}[x/\hat{\alpha}(\mathcal{P}')]$ be the PMC resulting from evaluation, denoted by $(S_e, s_{e_0}, s_{e_{suc}}, s_{e_{err}}, X_e, \mathbf{P}_e, T_e)$, and $\mathcal{P}_c = \mathcal{P}[x/\mathcal{P}']$ be the PMC obtained by composition, denoted by the tuple $(S_c, s_{c_0}, s_{c_{suc}}, s_{c_{err}}, X_c, \mathbf{P}_c, T_c)$. Figures A.1a and A.1b represent these PMCs and serve as a visual aid to the proof.




(a) $\mathcal{P}_e = \mathcal{P}[x/\hat{\alpha}(\mathcal{P}')]$

(b) $\mathcal{P}_c = \mathcal{P}[x/\mathcal{P}']$

(c) $\mathcal{P}_c$ after eliminating states $s' \in S' \setminus \mathit{interface}(\mathcal{P}')$

(d) $\mathcal{P}_c$ after eliminating all states $s' \in S'$

Figure A.1: Generic PMCs in Lemma 13

Since $\hat{\alpha}$ computes the probabilistic reachability property, we base this proof on the algorithm by Hahn et al. [41]. This algorithm consists of successive eliminations of states, with the transition probability matrix being updated at each step. A useful property,

which Hahn et al. use to prove that the algorithm is sound, is that the probability of reaching the target states in the input PMC is an invariant, that is, it remains the same throughout elimination steps.

Let us apply the algorithm by Hahn et al. [41] to $\mathcal{P}_c$. For brevity, we show the composition via a single slot. In the case where more slots exist, the following argument can be applied sequentially to each slot and corresponding renaming of $\mathcal{P}'$.

Since the order in which states are eliminated is not fixed, we first eliminate states $s' \in S' \setminus \mathit{interface}(\mathcal{P}')$. The intermediate PMC at this point is given by Figure A.1c. These eliminations are restricted to states in $S'$, because the only transitions in $\mathbf{P}_c$ between states in $S$ and states in $S'$ are the ones connecting interface and slot (by construction—see Definition 34).

Now, we eliminate the interface states. Performing a single step of the algorithm by Hahn et al. (Definition 4), we eliminate $s'_0$ and update $\mathbf{P}_c$ so that

$$\begin{aligned}
\mathbf{P}_c(s_{x_0}, s'_{suc}) &= \mathbf{P}_c(s_{x_0}, s'_{suc}) + \mathbf{P}_c(s_{x_0}, s'_0) \cdot \frac{1}{1 - \mathbf{P}_c(s'_0, s'_0)} \cdot \mathbf{P}_c(s'_0, s'_{suc}) \\
&= 0 + 1 \cdot \frac{1}{1-0} \cdot Pr^{\mathcal{P}'}(s'_0, s'_{suc}) \\
&= Pr^{\mathcal{P}'}(s'_0, s'_{suc})
\end{aligned}$$

Similarly, $\mathbf{P}_c(s_{x_0}, s'_{err}) = Pr^{\mathcal{P}'}(s'_0, s'_{err})$. Repeating these steps for $s'_{suc}$ and $s'_{err}$, $\mathbf{P}_c$ is updated to have $\mathbf{P}_c(s_{x_0}, s_{x_{suc}}) = Pr^{\mathcal{P}'}(s'_0, s'_{suc})$ and $\mathbf{P}_c(s_{x_0}, s_{x_{err}}) = Pr^{\mathcal{P}'}(s'_0, s'_{err})$ (see Figure A.1d).

At this stage, all states $s' \in S'$ have been eliminated, so that $S_c = S = S_e$. Furthermore, for all $s_1, s_2 \in S \setminus \mathit{slotStates}^{\mathcal{P}}(x)$, the transition probability matrices are such that $\mathbf{P}_c(s_1, s_2) = \mathbf{P}(s_1, s_2) = \mathbf{P}_e(s_1, s_2)$ (Definition 34). Thus, the only difference between $\mathcal{P}_c$ and $\mathcal{P}_e$ are the transitions for slot states: $(s_{x_0}, s_{x_{suc}})$ and $(s_{x_0}, s_{x_{err}})$.

For the "success" slot, $\mathbf{P}_c(s_{x_0}, s_{x_{suc}}) = Pr^{\mathcal{P}'}(s'_0, s'_{suc})$, which is *syntactically* equal to $\mathbf{P}_e(s_{x_0}, s_{x_{suc}})$. So, we must prove that the "error" transitions, $\mathbf{P}_c(s_{x_0}, s_{x_{err}})$ and $\mathbf{P}_e(s_{x_0}, s_{x_{err}})$, are extensionally equal. But $s'_{suc}$ and $s'_{err}$ are the only two bottom strongly connected components of the underlying digraph of $\mathcal{P}'$ (Definition 10). Thus, by Theorem 10.27 of Baier and Katoen [7], $Pr^{\mathcal{P}'_u}(s'_0, s'_{suc}) + Pr^{\mathcal{P}'_u}(s'_0, s'_{err}) = 1$, where $\mathcal{P}'_u$ is the DTMC obtained by applying some well-defined evaluation $u$ to $\mathcal{P}'$. Since the choice of $u$ is arbitrary, $\mathbf{P}_c(s_{x_0}, s_{x_{err}})$ is extensionally equal to $\mathbf{P}_e(s_{x_0}, s_{x_{err}})$.

This means that, at the current point of application of the probabilistic reachability algorithm to $\mathcal{P}_c$, $\mathbf{P}_c = \mathbf{P}_e$. $\mathcal{P}_e$ and the partially analyzed $\mathcal{P}_c$ have the same probability of reaching the target state $s_{suc}$. Moreover, since the algorithm preserves this probability at each step, the probabilistic reachability in $\mathcal{P}_c$ is the same at this point as be-

fore the algorithm started, and will remain the same until the algorithm stops. Hence, $\hat{\alpha}(\mathcal{P}[x/\mathcal{P}']) = \hat{\alpha}(\mathcal{P}[x/\hat{\alpha}(\mathcal{P}')])$.

To extend this proof to the case where $\mathcal{P}$ has an arbitrary number of variables, we repeat the argument that the choice of states for elimination is arbitrary. Let us assume, as induction hypothesis, that the lemma holds for a PMC with $n$ variables. If $\mathcal{P}$ has $n+1$ variables, we apply the same reasoning as in the single-variable case for one of $\mathcal{P}$'s slots, $(s_{x_{n+1_0}}, s_{x_{n+1_{suc}}}, s_{x_{n+1_{err}}})$. After eliminating *only* the states corresponding to a composition at the given slot, we have the following extensional equalities: $\mathbf{P}_c(s_{x_{n+1_0}}, s_{x_{n+1_{suc}}}) = \mathbf{P}_e(s_{x_{n+1_0}}, s_{x_{n+1_{suc}}})$ and $\mathbf{P}_c(s_{x_{n+1_0}}, s_{x_{n+1_{err}}}) = \mathbf{P}_e(s_{x_{n+1_0}}, s_{x_{n+1_{err}}})$. Also, the resulting PMC $\mathbf{P}_c$ has $n$ remaining slots, one for each variable. By the induction hypothesis, after eliminating the states corresponding to all compositions in $\mathbf{P}_c$, we have that $\mathbf{P}_c$ and $\mathbf{P}_e$ are extensionally equal. Hence, $\hat{\alpha}(\mathcal{P}[x_1/\mathcal{P}_1, \ldots, x_{n+1}/\mathcal{P}_{n+1}]) = \hat{\alpha}(\mathcal{P}[x_1/\hat{\alpha}(\mathcal{P}_1), \ldots, x_{n+1}/\hat{\alpha}(\mathcal{P}_{n+1})])$. □

Furthermore, since a composition of only DTMCs into a PMC yields another DTMC, both parametric and non-parametric model checking of this resulting chain (which has no variability) produce the same result. Thus, we have the following corollary of Lemma 13.

**Corollary 1** (r-equivalence of total composition with DTMCs and evaluation)**.** *Let $\mathcal{P}$ be a compositional PMC, $\mathcal{D}_1, \ldots, \mathcal{D}_k$ be DTMCs, and $X = \{x_1, \ldots, x_k\}$ be $\mathcal{P}$'s variables set. Then,*

$$\alpha(\mathcal{P}[x_1/\mathcal{D}_1, \ldots, x_k/\mathcal{D}_k]) = \alpha(\mathcal{P}[x_1/\alpha(\mathcal{D}_1), \ldots, x_k/\alpha(\mathcal{D}_k)])$$

Now we have the tools to prove that our feature-product-based analysis is sound. We recall Theorem 5:

**Theorem 5** (Soundness of feature-product-based analysis)**.** *Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$, for all configurations $c \in [\![FM]\!]$, it holds that*

$$\sigma(\hat{\alpha}(\mathcal{P}), w, c) = \alpha(\pi'(\mathcal{P}, w', c))$$

*or, alternatively,*

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

*where $\mathcal{P} \in \mathscr{P}$ and $w$ is the compositional evaluation factory (Definition 26) derived from the composition factory $w'$.*

*Complete proof.* We use well-founded induction. The base of the induction is when $\mathcal{P}$ is minimal with respect to $\prec$. In this case, $X = \emptyset$, so $\pi'(\mathcal{P}, w', c) = \mathcal{P}$, that is, $\alpha(\pi'(\mathcal{P}, w', c)) = \alpha(\mathcal{P})$. Likewise, $\hat{\alpha}(\mathcal{P}) = \alpha(\mathcal{P})$, so that $\sigma(\hat{\alpha}(\mathcal{P}), w, c) = \sigma(\alpha(\mathcal{P}), w, c) = \alpha(\mathcal{P})$. Thus, for the base case, $\sigma(\hat{\alpha}(\mathcal{P}), w, c) = \alpha(\pi'(\mathcal{P}, w', c))$.

We now have to prove that $\sigma(\hat{\alpha}(\mathcal{P}), w, c) = \alpha(\pi'(\mathcal{P}, w', c))$ for an arbitrary $\mathcal{P} \in \mathscr{P}$. Our induction hypothesis is that $\sigma(\hat{\alpha}(\mathcal{P}_i), w, c) = \alpha(\pi'(\mathcal{P}_i, w', c))$ for all $\mathcal{P}_i \in \mathscr{P}$ such that $\mathcal{P}_i \prec \mathcal{P}$. Thus, let $x_i = idt(\mathcal{P}_i)$, $i \in \{1, \ldots, k\}$. By Definition 22, we have:

$$\sigma(\hat{\alpha}(\mathcal{P}), w, c) = \hat{\alpha}(\mathcal{P})[x_1/w(c)(x_1), \ldots, x_k/w(c)(x_k)]$$

For each $x_i$, from the definition of the compositional evaluation factory $w$ (Definition 26),

$$\begin{aligned} w(c)(x_i) &= \begin{cases} \sigma(\hat{\alpha}(\mathcal{P}_i), w, c) & \text{if } p(x_i)(c) = 1 \\ 1 & \text{otherwise} \end{cases} \\ &= \begin{cases} \alpha(\pi'(\mathcal{P}_i, w', c)) & \text{if } p(x_i)(c) = 1 \\ 1 & \text{otherwise} \end{cases} && \text{(by induction hypothesis)} \end{aligned}$$

But, from the definition of the composition factory $w'$ (Definition 18),

$$\begin{aligned} w'(c)(x_i) &= \begin{cases} \mathcal{P}_i[X_i/w'(c)] & \text{if } p(x_i)(c) = 1 \\ \mathcal{P}_\perp & \text{otherwise} \end{cases} \\ &= \begin{cases} \pi'(\mathcal{P}_i, w', c)) & \text{if } p(x_i)(c) = 1 \\ \mathcal{P}_\perp & \text{otherwise} \end{cases} && \text{(Definition 19)} \end{aligned}$$

Applying $\alpha$ to both sides,

$$\alpha(w'(c)(x_i)) = \begin{cases} \alpha(\pi'(\mathcal{P}_i, w', c)) & \text{if } p(x_i)(c) = 1 \\ \alpha(\mathcal{P}_\perp) & \text{otherwise} \end{cases}$$

and, since $\alpha(\mathcal{P}_\perp) = 1$,

$$\begin{aligned} &= \begin{cases} \alpha(\pi'(\mathcal{P}_i, w', c)) & \text{if } p(x_i)(c) = 1 \\ 1 & \text{otherwise} \end{cases} \\ &= w(c)(x_i) \end{aligned}$$

Thus, $w(c)(x_i) = \alpha(w'(c)(x_i))$ and we have the following:

$$\begin{aligned} \sigma(\hat{\alpha}(\mathcal{P}), w, c) &= \hat{\alpha}(\mathcal{P})[x_1/w(c)(x_1), \ldots, x_k/w(c)(x_k)] \\ &= \hat{\alpha}(\mathcal{P})[x_1/\alpha(w'(c)(x_1)), \ldots, x_k/\alpha(w'(c)(x_k))] \\ &= \alpha(\mathcal{P}[x_1/\alpha(w'(c)(x_1)), \ldots, x_k/\alpha(w'(c)(x_k))]) && \text{(Lemma 3)} \\ &= \alpha(\mathcal{P}[x_1/w'(c)(x_1), \ldots, x_k/w'(c)(x_k)]) && \text{(Corollary 1)} \\ &= \alpha(\pi'(\mathcal{P}, w', c)) && \text{(Definition 19)} \end{aligned}$$

□

## A.4 Lifting Lemmas

This appendix covers details of lemmas related to lifting of expressions and of compositional evaluation factories.

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

*Complete proof.* The proof is by structural induction on the expression $\varepsilon$. The base cases are constant expressions and single variables:

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

  In this case, $\hat{\varepsilon} = \hat{c}$. Since $\varepsilon$ 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: $\hat{\varepsilon}[x/A](\bar{b}) = A(\bar{b}) = \varepsilon[x/A(\bar{b})]$.

Now we have to prove the statement holds for $\varepsilon = \varepsilon_1 \odot \varepsilon_2$ (where $\odot \in \{+, -, \times, \div\}$) and for $\varepsilon = \varepsilon_1^i$ (where $i \in \mathbb{N}$). As induction hypothesis, assume that the following holds for the expressions $\varepsilon_1$ and $\varepsilon_2$:

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

Let $u : X \to (\mathbb{B}^k \to \mathbb{R})$ be a lifted evaluation such that $u(x_i) = A_i$ is an ADD. We then have the following:

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

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

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

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

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

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

□

The soundness of lifted compositional evaluation factories is now presented in its complete form. First, we recall the corresponding lemma's statement.

**Lemma 6** (Soundness of lifted compositional evaluation factory). *Given a compositional model $(\mathscr{P}, \prec, I, idt, p, w', FM)$ and the compositional evaluation factory $w$, derived from the composition factory $w'$ (Definition 26), for all $x \in I$ and all $c \in [\![FM]\!]$ it holds that*

$$\varphi(x)(c) = w(c)(x)$$

*Complete proof.* If $\mathcal{P} \in \mathscr{P}$ is such that $idt(\mathcal{P}) = x$, then

$$\begin{aligned}\varphi(x)(c) &= \texttt{ITE}(\hat{p}(x), \widehat{\hat{\alpha}(\mathcal{P})}[X/\varphi], \hat{\mathbf{1}})(c) \\ &= \begin{cases} \widehat{\hat{\alpha}(\mathcal{P})}[X/\varphi](c) & \text{if } \hat{p}(x)(c) \neq 0 \\ \hat{\mathbf{1}}(c) & \text{if } \hat{p}(x)(c) = 0 \end{cases}\end{aligned}$$

By Lemma 4, $\widehat{\hat{\alpha}(\mathcal{P})}[X/\varphi](c) = \hat{\alpha}(\mathcal{P})[x_1/\varphi(x_1)(c), \ldots, x_k/\varphi(x_k)(c)]$. Also, $\forall_{c \in [\![FM]\!]} \cdot \hat{\mathbf{1}}(c) = 1$. Thus,

$$\varphi(x)(c) = \begin{cases} \hat{\alpha}(\mathcal{P})[x_1/\varphi(x_1)(c), \ldots, x_k/\varphi(x_k)(c)] & \text{if } \hat{p}(x)(c) \neq 0 \\ 1 & \text{if } \hat{p}(x)(c) = 0 \end{cases} \tag{A.1}$$

On the other hand, $w$ is defined (Definition 26) as

$$w(c)(x) = \begin{cases} [\![\hat{\alpha}(\mathcal{P})]\!]_c^w & \text{if } p(x)(c) \neq 0 \\ 1 & \text{if } p(x)(c) = 0 \end{cases}$$

Expanding the definition of $[\![\hat{\alpha}(\mathcal{P})]\!]_c^w$, we have

$$w(c)(x) = \begin{cases} \hat{\alpha}(\mathcal{P})[x_1/w(c)(x_1), \ldots, x_k/w(c)(x_k)] & \text{if } p(x)(c) \neq 0 \\ 1 & \text{if } p(x)(c) = 0 \end{cases} \tag{A.2}$$

Since $\hat{p}(x)(c) = p(x)(c)$, we compare corresponding cases in the Equations (A.1) and (A.2). The cases in which $p(x)(c) = 0$ are trivially equal. Otherwise, we use well-founded induction.

The base of our induction are minimal PMCs. A minimal PMC $\mathcal{P}$ has no variables ($X = \emptyset$), so $\hat{\alpha}(\mathcal{P})[X/u] = \alpha(\mathcal{P})$ for any evaluation $u$. Since $w(c)$ is an evaluation, and considering $\varphi(x)(c)$ takes a variable $x$ to a Real number (thus, also being an evaluation), we have that $\widehat{\hat{\alpha}(\mathcal{P})}[X/\varphi](c) = \hat{\alpha}(\mathcal{P})[X/w(c)]$ in this case. For non-minimal PMCs, assume, as induction hypothesis, that $\widehat{\hat{\alpha}(\mathcal{P}_j)}[X_j/\varphi](c) = \hat{\alpha}(\mathcal{P}_j)[X_j/w(c)]$ for all $\mathcal{P}_j \prec \mathcal{P}$, where $j \in \{1, \ldots, k\}$. Then, for any $x_j \in X$,

$$\varphi(x_j)(c) = \begin{cases} \widehat{\hat{\alpha}(\mathcal{P}_j)}[X_j/\varphi](c) & \text{if } \hat{p}(x_j)(c) \neq 0 \\ 1 & \text{if } \hat{p}(x_j)(c) = 0 \end{cases} \tag{A.3}$$

$$w(c)(x_j) = \begin{cases} \hat{\alpha}(\mathcal{P}_j)[X_j/w(c)] & \text{if } p(x_j)(c) \neq 0 \\ 1 & \text{if } p(x_j)(c) = 0 \end{cases} \tag{A.4}$$

However, the induction hypothesis implies the right-hand sides of the Equations (A.3) and (A.4) are equal. Thus, $\varphi(x_j)(c) = w(c)(x_j)$ for all $x_j \in X$, which means

$$\hat{\alpha}(\mathcal{P})[x_1/\varphi(x_1)(c), \ldots, x_k/\varphi(x_k)(c)] = \hat{\alpha}(\mathcal{P})[x_1/w(c)(x_1), \ldots, x_k/w(c)(x_k)]$$

and, by well-founded induction, the cases where $p(x)(c) = 1$ in the Equations (A.1) and (A.2) are also equal. Hence, $\varphi(x)(c) = w(c)(x)$. □

## A.5 Variability Encoding

This appendix deals with formal definitions and complete proofs related to variability encoding of PMCs and of rational expressions.

### A.5.1 Variability Encoding of PMCs

We start by formally defining the `ITE` operator for PMCs, which was only presented as an intuition in the main body of the paper.

**Definition 35** (`ITE` operator for PMCs)**.** Given two compositional PMCs, $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$ and $\mathcal{P}' = (S', s'_0, s'_{suc}, s'_{err}, X', \mathbf{P}', T')$, and a variable $x \notin X \cup X'$, the *if-then-else* operator for PMCs is defined as

$$\texttt{ITE}(x, \mathcal{P}, \mathcal{P}') = \mathcal{P}''$$

where $\mathcal{P}'' = (S'', s''_0, s''_{suc}, s''_{err}, X'', \mathbf{P}'', T'')$ is a compositional PMC such that:

- $S'' = S \cup S' \cup \{s''_0, s''_{suc}, s''_{err}\}$
- The state $s''_0$ is the new initial one, $s''_{suc}$ is the new success state, and $s''_{err}$ is the new error state.
- $X'' = X \cup X' \cup \{x\}$
- $T'' = \{s''_{suc}\}$
- $\mathbf{P}''$ is such that:
  - $\mathbf{P}''(s''_0, s_0) = x$
  - $\mathbf{P}''(s''_0, s'_0) = 1 - x$
  - $\mathbf{P}''(s_{suc}, s''_{suc}) = \mathbf{P}''(s'_{suc}, s''_{suc}) = \mathbf{P}''(s''_{suc}, s''_{suc}) = 1$
  - $\mathbf{P}''(s_{suc}, s_{suc}) = \mathbf{P}''(s'_{suc}, s'_{suc}) = 0$

- $\mathbf{P}''(s_{err}, s''_{err}) = \mathbf{P}''(s'_{err}, s''_{err}) = \mathbf{P}''(s''_{err}, s''_{err}) = 1$
- $\mathbf{P}''(s_{err}, s_{err}) = \mathbf{P}''(s'_{err}, s'_{err}) = 0$
- For all remaining combinations of $s_1, s_2 \in S''$:

$$\mathbf{P}''(s_1, s_2) = \begin{cases} \mathbf{P}(s_1, s_2) & \text{if } s_1, s_2 \in S \\ \mathbf{P}'(s_1, s_2) & \text{if } s_1, s_2 \in S' \\ 0 & \text{otherwise} \end{cases}$$

This `ITE` operator is mainly useful because of its r-equivalence property. We recall Lemma 7 and present its complete proof:

**Lemma 7** (r-equivalence for `ITE`)**.** *Given two compositional PMCs, $\mathcal{P} = (S, s_0, s_{suc}, s_{err}, X, \mathbf{P}, T)$ and $\mathcal{P}' = (S', s'_0, s'_{suc}, s'_{err}, X', \mathbf{P}', T')$, and a variable $x \notin X \cup X'$, let $\mathcal{P}'' = \mathtt{ITE}(x, \mathcal{P}, \mathcal{P}')$. If $(\mathcal{P}'', p, w, FM)$ is an annotative model with $\mathcal{P}''$ as its underlying PMC*[1]*, where $p$, $w$, and $FM$ are arbitrarily chosen, then, for every $c \in [\![FM]\!]$,*

$$\alpha([\![\mathtt{ITE}(x, \mathcal{P}, \mathcal{P}')]\!]^w_c) = \begin{cases} \alpha([\![\mathcal{P}]\!]^w_c) & \text{if } p(x)(c) = 1 \\ \alpha([\![\mathcal{P}']\!]^w_c) & \text{otherwise} \end{cases}$$

*Complete proof.* We are interested in computing the probability of reaching $s''_{suc}$ from $s''_0$ in $\mathcal{P}'' = \mathtt{ITE}(x, \mathcal{P}, \mathcal{P}')$ under evaluation $w(c)$. In $\mathcal{P}''$, $s''_0 \neq s''_{suc}$ and $s''_{suc}$ is reachable from $s''_0$ (since $s''_{suc}$ is, by definition, reachable from $s_{suc}$ and $s'_{suc}$). Hence, the reachability of $s''_{suc}$ from $s''_0$ satisfies Property 1, by which the probability of reaching state $s_2$ from state $s_1$ in a DTMC $\mathcal{D} = (S, s_0, \mathbf{P}, T)$ is given by

$$Pr^{\mathcal{D}}(s_1, s_2) = \sum_{s' \in S} \mathbf{P}(s_1, s') \cdot Pr^{\mathcal{D}}(s', s_2)$$

[1] By Definition 10, any compositional PMC is also an annotative PMC (Definition 5). Thus, a compositional PMC can be the underlying PMC of an annotative model.

By Definition 35, $\mathbf{P}''(s''_0, s_0) = x$, $\mathbf{P}''(s''_0, s'_0) = 1 - x$, and $\mathbf{P}''(s''_0, s'') = 0$ for all other $s'' \in S''$. Thus,

$$\begin{aligned}
\alpha(\llbracket \mathcal{P}'' \rrbracket_c^w) &= Pr^{\mathcal{P}''_{w(c)}}(s''_0, s''_{suc}) \\
&= \sum_{s'' \in S''} \mathbf{P}''_{w(c)}(s''_0, s'') \cdot Pr^{\mathcal{P}''_{w(c)}}(s'', s''_{suc}) \\
&= \sum_{s'' \in S'' \setminus \{s''_{suc}\}} \mathbf{P}''_{w(c)}(s''_0, s'') \cdot Pr^{\mathcal{P}''_{w(c)}}(s'', s''_{suc}) + \mathbf{P}''_{w(c)}(s''_0, s''_{suc}) \\
&= \sum_{s'' \in S'' \setminus \{s''_{suc}\}} \mathbf{P}''_{w(c)}(s''_0, s'') \cdot Pr^{\mathcal{P}''_{w(c)}}(s'', s''_{suc}) + 0 \\
&= \mathbf{P}''_{w(c)}(s''_0, s_0) \cdot Pr^{\mathcal{P}''_{w(c)}}(s_0, s''_{suc}) + \mathbf{P}''_{w(c)}(s''_0, s'_0) \cdot Pr^{\mathcal{P}''_{w(c)}}(s'_0, s''_{suc}) \\
&= w(c)(x) \cdot Pr^{\mathcal{P}''_{w(c)}}(s_0, s''_{suc}) + (1 - w(c)(x)) \cdot Pr^{\mathcal{P}''_{w(c)}}(s'_0, s''_{suc})
\end{aligned}$$

Since $w(c)(x)$ equals 1 if $p(x)(c) = 1$ and 0 otherwise (Definition 8),

$$\alpha(\llbracket \mathcal{P}'' \rrbracket_c^w) = \begin{cases} Pr^{\mathcal{P}''_{w(c)}}(s_0, s''_{suc}) & \text{if } p(x)(c) = 1 \\ Pr^{\mathcal{P}''_{w(c)}}(s'_0, s''_{suc}) & \text{otherwise} \end{cases}$$

But, since $s_0 \in S$ and the only state in $S$ that can reach $s''_{suc}$ is $s_{suc}$ (Definition 35), the probability of reaching $s''_{suc}$ from $s_0$ is the probability of reaching $s_{suc}$ from $s_0$ multiplied by the transition probability from $s_{suc}$ to $s''_{suc}$:

$$\begin{aligned}
Pr^{\mathcal{P}''_{w(c)}}(s_0, s''_{suc}) &= Pr^{\mathcal{P}''_{w(c)}}(s_0, s_{suc}) \cdot \mathbf{P}''_{w(c)}(s_{suc}, s''_{suc}) \\
&= Pr^{\mathcal{P}_{w(c)}}(s_0, s_{suc}) \cdot 1 \\
&= Pr^{\mathcal{P}_{w(c)}}(s_0, s_{suc}) \\
&= \alpha(\llbracket \mathcal{P} \rrbracket_c^w)
\end{aligned}$$

Similar reasoning applied to $S'$ leads to $Pr^{\mathcal{P}''_{w(c)}}(s'_0, s''_{suc}) = \alpha(\llbracket \mathcal{P}' \rrbracket_c)$. Hence,

$$\alpha(\llbracket \mathcal{P}'' \rrbracket_c^w) = \begin{cases} \alpha(\llbracket \mathcal{P} \rrbracket_c^w) & \text{if } p(x)(c) = 1 \\ \alpha(\llbracket \mathcal{P}' \rrbracket_c^w) & \text{otherwise} \end{cases}$$

□

The above lemma establishes the `ITE` operator has the effect of alternating behaviors if the resulting PMC is evaluated by replacing the switching variable $x$ with 0 or 1. However, the PMC operands of `ITE` are part of a compositional model, so their own variables are interpreted as placeholders to be used during composition, instead (see Section 3.1.2). To

cope with this mismatch, we only use the ITE operator with PMCs that are either plain DTMCs or that result themselves from variability encoding.

The resulting theorem stating the soundness of this variability encoding for PMCs is recalled and proved next.

**Theorem 8** (r-equivalence of variability encoding and derivation by composition)**.** *Given a compositional model* $(\mathscr{P}, \prec, I, idt, p, w', FM)$ *and* $\mathcal{P} \in \mathscr{P}$*, let* $(\gamma(\mathcal{P}), p, w, FM)$ *be its variability-encoded annotative model. Then, for all* $c \in [\![FM]\!]$*,*

$$\alpha([\![\gamma(\mathcal{P})]\!]_c^w) = \alpha(\pi'(\mathcal{P}, w', c))$$

*Complete proof.* We use well-founded induction. For minimal PMCs (base of induction), $\gamma(\mathcal{P}) = \mathcal{P}$, so $[\![\gamma(\mathcal{P})]\!]_c^w = \mathcal{P}$. Likewise, $\pi'(\mathcal{P}, w', c) = \mathcal{P}$, so the proposition holds trivially.

As induction hypothesis, we have that $\alpha([\![\gamma(\mathcal{P}_i)]\!]_c^w) = \alpha(\pi'(\mathcal{P}_i, w', c))$ for all $\mathcal{P}_i \in \mathscr{P}$ such that $\mathcal{P}_i \prec \mathcal{P}$. For brevity, in the following equations, we use $\Lambda_i$ to denote $\texttt{ITE}(x_i, \gamma(\mathcal{P}_i), \mathcal{P}_\perp)$.

$$\begin{aligned}
\alpha([\![\gamma(\mathcal{P})]\!]_c^w) &= [\![\hat{\alpha}(\gamma(\mathcal{P}))]\!]_c^w && \text{(Theorem 1)}\\
&= [\![\hat{\alpha}(\mathcal{P}[x_1/\Lambda_1, \ldots, x_k/\Lambda_k])]\!]_c^w && \text{(Definition 28)}\\
&= [\![\hat{\alpha}(\mathcal{P}[x_1/\hat{\alpha}(\Lambda_1), \ldots, x_k/\hat{\alpha}(\Lambda_k)])]\!]_c^w && \text{(Lemma 13)}\\
&= [\![\hat{\alpha}(\mathcal{P})[x_1/\hat{\alpha}(\Lambda_1), \ldots, x_k/\hat{\alpha}(\Lambda_k)]]\!]_c^w && \text{(Lemma 3)}\\
&= \hat{\alpha}(\mathcal{P})[x_1/\hat{\alpha}(\Lambda_1), \ldots, x_k/\hat{\alpha}(\Lambda_k)][X/w(c)] && \text{(Definition 22)}\\
&= \hat{\alpha}(\mathcal{P})[x_1/\hat{\alpha}(\Lambda_1)[X/w(c)], \ldots, \\
&\qquad \ldots, x_k/\hat{\alpha}(\Lambda_k)[X/w(c)]] && \text{(Equation (2.1))}\\
&= \hat{\alpha}(\mathcal{P})[x_1/[\![\hat{\alpha}(\Lambda_1)]\!]_c^w, \ldots, x_k/[\![\hat{\alpha}(\Lambda_k)]\!]_c^w] && \text{(Definition 22)}\\
&= \hat{\alpha}(\mathcal{P})[x_1/\alpha([\![\Lambda_1]\!]_c^w), \ldots, x_k/\alpha([\![\Lambda_k]\!]_c^w)] && \text{(Theorem 1)}\\
&= \alpha(\mathcal{P}[x_1/\alpha([\![\Lambda_1]\!]_c^w), \ldots, x_k/\alpha([\![\Lambda_k]\!]_c^w)]) && \text{(Lemma 3)}
\end{aligned}$$

leaving us with the following partial result:

$$\alpha([\![\gamma(\mathcal{P})]\!]_c^w) = \alpha(\mathcal{P}[x_1/\alpha([\![\Lambda_1]\!]_c^w), \ldots, x_k/\alpha([\![\Lambda_k]\!]_c^w)]) \tag{A.5}$$

Each variable substitution expands to two different cases, corresponding to whether $c$ satisfies the presence condition associated with $x_i$ or not. Let us examine the substitution

for a given $x_i$:

$$\begin{aligned}
\alpha(\llbracket \Lambda_i \rrbracket_c^w) &= \alpha(\llbracket \mathtt{ITE}(x_i, \gamma(\mathcal{P}_i), \mathcal{P}_\perp) \rrbracket_c^w) \\
&= \begin{cases} \alpha(\llbracket \gamma(\mathcal{P}_i) \rrbracket_c^w) & \text{if } p(x_i)(c) = 1 \\ \alpha(\llbracket \mathcal{P}_\perp \rrbracket_c^w) & \text{otherwise} \end{cases} && \text{(Lemma 7)} \\
&= \begin{cases} \alpha(\pi'(\mathcal{P}_i, w', c)) & \text{if } p(x_i)(c) = 1 \\ \alpha(\llbracket \mathcal{P}_\perp \rrbracket_c^{w'}) & \text{otherwise} \end{cases} && \text{(by induction hypothesis)} \\
&= \alpha(w'(c)(x_i)) && \text{(Definitions 18 and 19)}
\end{aligned}$$

that is,

$$\alpha(\llbracket \Lambda_i \rrbracket_c^w) = \alpha(w'(c)(x_i)) \tag{A.6}$$

Hence, we can substitute Equation (A.6) into Equation (A.5):

$$\begin{aligned}
\alpha(\llbracket \gamma(\mathcal{P}) \rrbracket_c^w) &= \alpha(\mathcal{P}[x_1/\alpha(\llbracket \Lambda_1 \rrbracket_c^w), \ldots, x_k/\alpha(\llbracket \Lambda_k \rrbracket_c^w)]) && \text{(Equation (A.5))} \\
&= \alpha(\mathcal{P}[x_1/\alpha(w'(c)(x_1)), \ldots, x_k/\alpha(w'(c)(x_k))]) && \text{(Equation (A.6))} \\
&= \alpha(\mathcal{P}[x_1/w'(c)(x_1), \ldots, x_k/w'(c)(x_k)]) && \text{(Corollary 1)} \\
&= \alpha(\pi'(\mathcal{P}, w', c)) && \text{(Definition 19)}
\end{aligned}$$

□

### A.5.2 Variability Encoding of Expressions

We start by proving that the `ITE` operator for expressions has the intended semantics. This result is expressed by Lemma 8, which we now recall.

**Lemma 8** (Extensional equality for expression `ITE`)**.** *Given two expressions $\varepsilon$ and $\varepsilon'$ over the sets $X$ and $X'$ of variables, respectively, and a variable $x$, let $X'' = X \cup X' \cup \{x\}$ and $u : X'' \to [0, 1]$ be an evaluation function such that $u(x) \in \mathbb{B}$. Then,*

$$\mathtt{ITE}(x, \varepsilon, \varepsilon')[X''/u] = \begin{cases} \varepsilon[X/u] & \textit{if } u(x) = 1 \\ \varepsilon'[X'/u] & \textit{if } u(x) = 0 \end{cases}$$

*Complete proof.* The proof is mainly algebraic. Expanding the definition of `ITE`, we have:

$$\begin{aligned}
\texttt{ITE}(x, \varepsilon, \varepsilon')[X''/u] &= (x \cdot \varepsilon + (1-x) \cdot \varepsilon')[X''/u] \\
&= (x \cdot \varepsilon)[X''/u] + ((1-x) \cdot \varepsilon')[X''/u] \\
&= x[X''/u] \cdot \varepsilon[X''/u] + (1-x)[X''/u] \cdot \varepsilon'[X''/u] \\
&= u(x) \cdot \varepsilon[X''/u] + (1-u(x)) \cdot \varepsilon'[X''/u] \\
&= \begin{cases} \varepsilon[X''/u] & \text{if } u(x) = 1 \\ \varepsilon'[X''/u] & \text{if } u(x) = 0 \end{cases}
\end{aligned}$$

which, considering that the sets of variables in $\varepsilon$ and $\varepsilon'$ are $X$ and $X'$, respectively, and that these sets are subsets of $X''$, leads to

$$\texttt{ITE}(x, \varepsilon, \varepsilon')[X''/u] = \begin{cases} \varepsilon[X/u] & \text{if } u(x) = 1 \\ \varepsilon'[X'/u] & \text{if } u(x) = 0 \end{cases}$$

□

Using this result and the definitions in the main body of the paper, we can prove that variability encoding for expressions is sound.

**Theorem 9** (Soundness of variability encoding for expressions)**.** *Given a compositional model* $(\mathscr{P}, \prec, I, idt, p, w', FM)$ *and* $\mathcal{P}, \mathcal{P}_1, \ldots, \mathcal{P}_k \in \mathscr{P}$ *such that* $\mathcal{P}_i \prec \mathcal{P}$ *and* $x_i = idt(\mathcal{P}_i)$ *for* $i \in \{1, \ldots, k\}$, *let* $\varepsilon = \hat{\alpha}(\mathcal{P})$. *Let also* $w$ *be the compositional evaluation factory derived from* $w'$ *(Definition 26) and* $w_p$ *be the annotative evaluation factory obtained from* $w$ *(Definition 32). Then, for all* $c \in [\![FM]\!]$ *it holds that*

$$\sigma(\gamma(\varepsilon), w_p, c) = \sigma(\varepsilon, w, c)$$

*Complete proof.* We use well-founded induction. For a minimal PMC $\mathcal{P}$ (base of induction), $\hat{\alpha}(\mathcal{P}) = \varepsilon$ has no variables. This way, $\gamma(\varepsilon) = \varepsilon$ and $\sigma(\varepsilon, u) = \varepsilon$ for any evaluation $u$. Thus, both sides of the equality evaluate to $\varepsilon$ and the proposition holds trivially.

As induction hypothesis, we have that $\sigma(\gamma(\varepsilon_i), w_p, c) = \sigma(\varepsilon_i, w, c)$ for all $\varepsilon_i = \hat{\alpha}(\mathcal{P}_i)$ such that $\mathcal{P}_i \prec \mathcal{P}$. For brevity, we use $\Lambda_i$ to denote $\texttt{ITE}(x_i, \gamma(\varepsilon_i), \mathbf{1})$ in the following equations.

$$\begin{aligned}
\sigma(\gamma(\varepsilon), w_p, c) &= \sigma(\varepsilon[x_1/\Lambda_1, \ldots, x_k/\Lambda_k], w_p, c) && \text{(Definition 31)} \\
&= \varepsilon[x_1/\Lambda_1, \ldots, x_k/\Lambda_k][X/w_p(c)] && \text{(Definition 22)} \\
&= \varepsilon[x_1/\Lambda_1[X/w_p(c)], \ldots, x_k/\Lambda_k[X/w_p(c)]] && \text{(Equation (2.1))}
\end{aligned}$$

yielding the following equation:

$$\sigma(\gamma(\varepsilon), w_p, c) = \varepsilon[x_1/\Lambda_1[X/w_p(c)], \ldots, x_k/\Lambda_k[X/w_p(c)]] \tag{A.7}$$

Each variable substitution expands to two different cases, corresponding to whether $c$ satisfies the presence condition associated with $x_i$ or not. Let us examine the substitution for a given $x_i$:

$$\begin{aligned}
\Lambda_i[X/w_p(c)] &= \texttt{ITE}(x_i, \gamma(\varepsilon_i), \mathbf{1})[X/w_p(c)] \\
&= \begin{cases} \gamma(\varepsilon_i)[X/w_p(c)] & \text{if } p(x_i)(c) = 1 \ (w_p(c)(x_i) = 1) \text{ (Lemma 8)} \\ \mathbf{1}[X/w_p(c)] & \text{otherwise } (w_p(c)(x_i) = 0) \end{cases} \\
&= \begin{cases} \sigma(\gamma(\varepsilon_i), w_p, c) & \text{if } p(x_i)(c) = 1 \\ 1 & \text{otherwise} \end{cases} && \text{(Definition 22)} \\
&= \begin{cases} \sigma(\varepsilon_i, w, c) & \text{if } p(x_i)(c) = 1 \\ 1 & \text{otherwise} \end{cases} && \text{(by induction hypothesis)} \\
&= w(c)(x_i) && \text{(Definition 26)}
\end{aligned}$$

that is,

$$\Lambda_i[X/w_p(c)] = w(c)(x_i) \tag{A.8}$$

Hence, substituting Equation (A.8) into Equation (A.7), we have

$$\begin{aligned}
\sigma(\gamma(\varepsilon), w_p, c) &= \varepsilon[x_1/\Lambda_1[X/w_p(c)], \ldots, x_k/\Lambda_k[X/w_p(c)]] && \text{(Equation (A.7))} \\
&= \varepsilon[x_1/w(c)(x_1), \ldots, x_k/w(c)(x_k)] && \text{(Equation (A.8))} \\
&= \varepsilon[X/w(c)] \\
&= \sigma(\varepsilon, w, c) && \text{(Definition 22)}
\end{aligned}$$

□

# Appendix B

# Probabilistic Models

This appendix presents the probabilistic models of the beverage machine product line example (Section 3.1) in their entirety. Figure B.1 contains the annotative model, and the compositional model is depicted by Figure B.2.

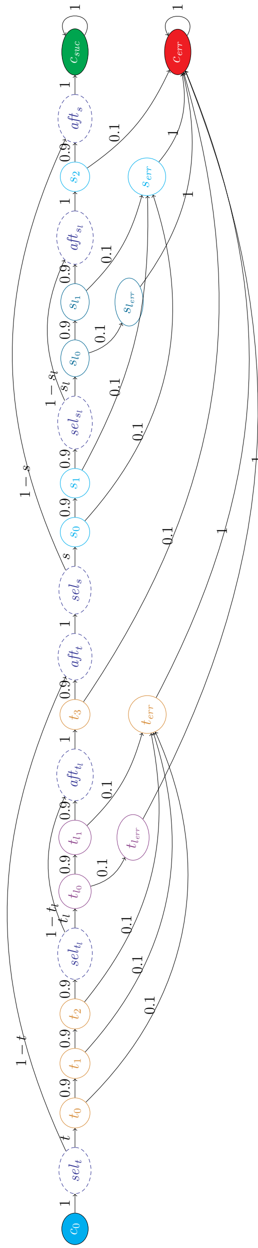
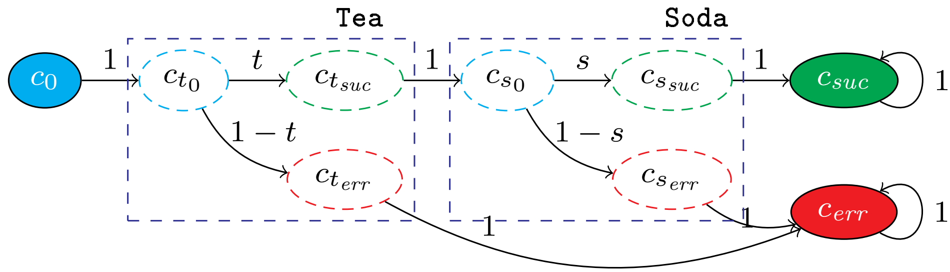
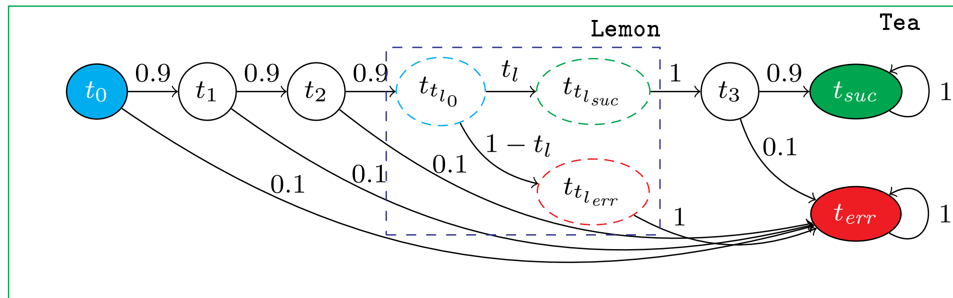


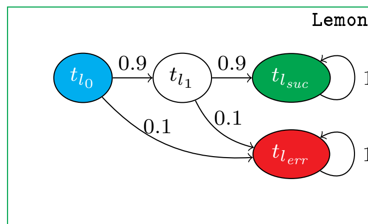
Figure B.1: Complete annotative PMC for the vending machine



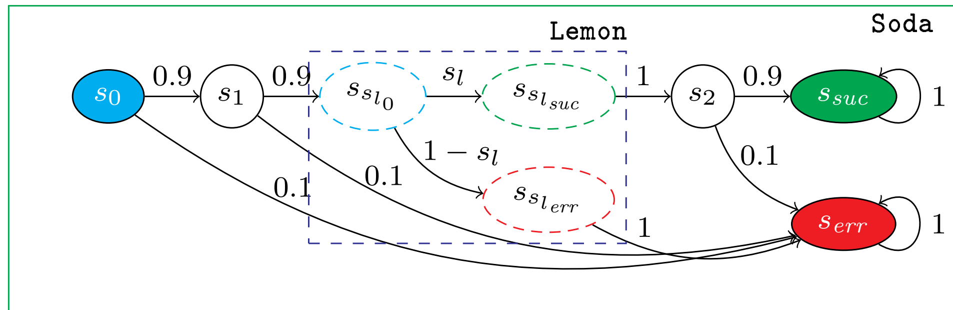

(a) Top-level compositional PMC for the vending machine (common behavior and main variation points)



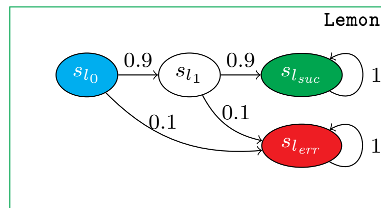

(b) Compositional PMC for the behavior of serving tea




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




(d) Compositional PMC for the behavior of serving soda




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

Figure B.2: Compositional PMCs for the vending machine

# Appendix C

# Mechanization Mapping

This appendix presents the complete description of our PVS theories (Table C.1) and the interdependencies between them (Figure C.1). We also present the correspondence between elements of our manual specification (Chapter 3) and its mechanization in PVS (Chapter 4) in Table C.2. In this table, each entry for an element of the original theory is linked to its definition in Chapter 3, whereas the corresponding PVS definition is also an hyperlink to its specific line in the Github repository (`https://github.com/thiagomael/rome-specs`). Last, we present the complete data set representing the count of proof commands (Tables C.3 and C.4) and the distribution of theory elements throughout the PVS mechanization (Table C.5).

Table C.1: Description of PVS theories

| **PVS Theory** | **Description** |
|---|---|
| `ADD_def` (ADD.pvs, line 1) | Definitions regarding ADDs with terminals of a parameterized type `T`. |
| `ADD_ops` (ADD.pvs, line 15) | Semantics of arithmetic operations with ADDs. |
| `real_ADD` (ADD.pvs, line 48) | Real-valued ADDs. |
| `annotative_expressions_evaluation` (annotative_expressions_evaluation.pvs, line 1) | Evaluation of lifted expressions resulting from an annotative model. |
| `annotative_PMC` (annotative_PMC.pvs, line 1) | Definition of annotative PMC and corresponding predicates. This theory also contains lemmas about the evaluation of such PMCs. |

Table C.1: Description of PVS theories (continued)

| PVS Theory | Description |
|---|---|
| annotative_reliability_models (annotative_reliability_models.pvs, line 1) | Definition of annotative reliability models and corresponding predicates. |
| compositional_expressions_evaluation (compositional_expressions_evaluation.pvs, line 1) | Evaluation of lifted expressions resulting from a compositional model. |
| compositional_PMC_order (compositional_PMC_order.pvs, line 1) | Well-founded order induced by the dependency relation in compositional reliability models. |
| compositional_PMC (compositional_PMC.pvs, line 1) | Definition of compositional PMC and corresponding predicates. This theory also contains the definition and the main lemmas about slots. |
| compositional_PMC_sets (compositional_PMC_sets.pvs, line 1) | Definitions regarding identifiers and variables of a finite set of PMCs. |
| compositional_reliability_models (compositional_reliability_models.pvs, line 1) | Definition of compositional reliability models and corresponding predicates. |
| compositional_factory_restrict (compositional_reliability_models.pvs, line 112) | Auxiliary definitions to restrict the higher-order compositional factories (composition factory and compositional evaluation factory). |
| compositional_reliability_models_lemmas (compositional_reliability_models.pvs, line 132) | Auxiliary lemmas regarding the exhaustion of variables by composition. |
| DTMC (DTMC.pvs, line 1) | DTMC and related concepts (paths, reachability, and reachability probability). This theory also presents lemmas about path probabilities and morphisms between DTMCs. |
| expression_lifting (expression_lifting.pvs, line 1) | Evaluation of rational expressions using ADDs. |

Table C.1: Description of PVS theories (continued)

| PVS Theory | Description |
| --- | --- |
| expressions_variability_encoding (expressions_variability_encoding.pvs, line 1) | Variability encoding of expressions. This theory provides both the domain-agnostic formalization of the if-then-else operator and the encoding of expressions that denote the reliability of compositional models. |
| finite_set_lemmas (finite_set_lemmas.pvs, line 1) | Lemmas about functional mappings between finite sets of different types. |
| finite_sets_aux (finite_set_lemmas.pvs, line 50) | Lemmas about finite sets of the same type and sums thereof. |
| infinite_set_lemmas (infinite_set_lemmas.pvs, line 1) | Lemmas about infinite sums over infinite sets. |
| list_aux (list_aux.pvs, line 1) | Lemmas about list properties, especially of lists without repeated elements (sets as lists). |
| list_map_aux (list_map_aux.pvs, line 6) | This theory is a means to resolve the mismatch of type parameters that arises from using the lemmas in more_map_props (which require the type of elements in the list to be *exactly* the same as the domain D of the mapped function). Here we relax this constraint to allow any supertype of D. |
| maybe_real (maybe_real.pvs, line 1) | Datatype for optional Real numbers and the corresponding arithmetic operators. This theory provides a mechanism to handle division by zero and the propagation of undefined results (similar to the behavior of the Maybe monad in the Haskell programming language). |

Table C.1: Description of PVS theories (continued)

| PVS Theory | Description |
|---|---|
| parametric_transition_matrices (parametric_transition_matrices.pvs, line 1) | Specialization of transition matrices for which the cells (transitions) are rational expressions. This theory also provides the notions of variables of a matrix and of matrix evaluation, as well as related lemmas. |
| PMC_composition (PMC_composition.pvs, line 1) | Definition of partial and total PMC composition, along with lemmas stating the properties of the results. |
| PMC (PMC.pvs, line 1) | Parametric DTMCs and related definitions, such as well-defined evaluations and state elimination. |
| PMC_renaming (PMC_renaming.pvs, line 2) | Renaming of PMCs, specifically designed to provide a disjoint union of the sets of states during PMC composition. |
| PMC_r_equivalence (PMC_r_equivalence.pvs, line 1) | Lemmas regarding r-equivalence of PMCs under state elimination. |
| PMC_variability_encoding (PMC_variability_encoding.pvs, line 1) | If-then-else operator for transition matrices and PMCs, along with the notions specific to variability encoding of compositional reliability models. |
| rational_expressions (rational_expressions.pvs, line 1) | Definitions and lemmas regarding a datatype-oriented view of rational expressions (prone to syntactic manipulation). |
| real_transition_matrices (real_transition_matrices.pvs, line 1) | Specialization of transition matrices whose cells (transitions) are probabilities and whose rows obey the stochastic property. |
| rome (rome.pvs, line 1) | Top-level theory, containing the definitions of our analysis strategies and corresponding soundness proofs. |

Table C.1: Description of PVS theories (continued)

| PVS Theory | Description |
|---|---|
| SPL_expression_evaluation (SPL_expression_evaluation.pvs, line 1) | Definition of expression evaluation using evaluation factories. |
| SPL (SPL.pvs, line 1) | Uninterpreted definitions of product-line semantics. |
| SPL_reliability (SPL_reliability.pvs, line 1) | Definition of evaluation factory and of evaluation-based derivation of DTMCs from either compositional or annotative PMCs. |
| states (states.pvs, line 1) | Semantics of states as an infinite type. |
| transition_matrices (transition_matrices.pvs, line 1) | Parameterized definition of transition matrices as a record type. |
| transition_matrices_map (transition_matrices.pvs, line 26) | Definition of function mappings over transition matrices. |
| transition_matrices_sum (transition_matrices.pvs, line 48) | Definition of finite sums of values in transition matrices. |
| variability_aware_expression_evaluation (variability_aware_expression_evaluation.pvs, line 1) | Lifting of evaluation factories and mapping between product line configurations and evaluations of ADD arguments. |
| well_founded_lemmas (well_founded_lemmas.pvs, line 1) | Auxiliary lemmas for well-founded relations over finite types. |

Figure C.1 shows the mechanized theories used in our work and the dependencies among them. In this figure, third-party theories (PVS prelude and NASA libraries) are depicted as blue rectangles. Ellipses represent the theories created in the scope of this work, according to the following color code:

- green denotes theories with additional lemmas for concepts that already exist in the third-party libraries (high reuse potential);
- yellow is used for theories with concepts that are new to PVS, but exist in the literature (medium to high reuse potential);
- white is used for theories that are specific to the user-oriented reliability analysis of product lines presented in this work.

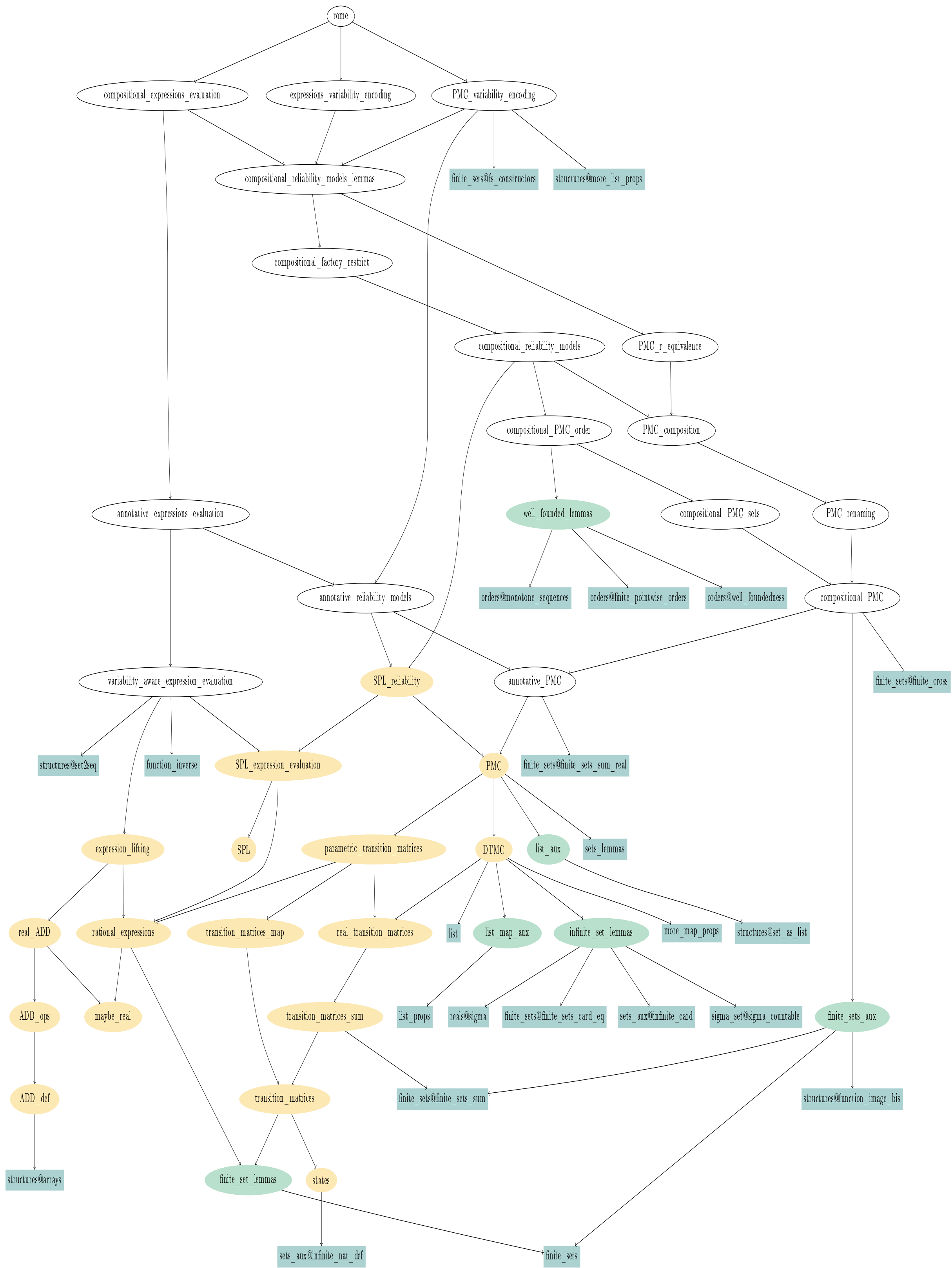



Figure C.1: Dependencies between PVS theories (direct dependencies that can be deduced from transitivity are filtered for readability)

Table C.2: Mapping between the manual and the mechanized specifications

| Manual | Mechanized |
|---|---|
| Property 1 (Reachability probability for DTMCs) | `reachability_probability_property`<br>(DTMC.pvs, line 41) — **AXIOM** |
| Definition 1 (Parametric Markov Chain) | `PMC`<br>(PMC.pvs, line 9) |
| Definition 2 (Expression evaluation) | `eval`<br>(rational_expressions.pvs, line 49) |
| Definition 3 (Well-defined evaluation) | `well_defined_evaluation`<br>(PMC.pvs, line 42) |
| Definition 4 (State elimination step) | `eliminate_state`<br>(PMC.pvs, line 195) |
| Lemma 1 (Parametric probabilistic reachability soundness) | `parametric_reachability_soundness`<br>(PMC.pvs, line 54) — **AXIOM** |
| Definition 5 (Annotative PMC) | `annotative_PMC`<br>(annotative_PMC.pvs, line 29) |
| Definition 6 (Presence function) | `presence_function`<br>(SPL_reliability.pvs, line 11) |
| Definition 7 (Evaluation factory) | `evaluation_factory`<br>(SPL_reliability.pvs, line 14) |
| Definition 8 (Annotative probabilistic model) | `annotative_reliability_model, interface`<br>(annotative_reliability_models.pvs, line 34) |
| Definition 9 (DTMC derivation) | `pi`<br>(SPL_reliability.pvs, line 23) |

Table C.2: Mapping between the manual and the mechanized specifications (continued)

| Manual | Mechanized |
|---|---|
| Lemma 2 (Evaluation well-definedness for annotative models) | `the_annotative_evaluation_factory_well_definedness` (annotative_reliability_models.pvs, line 27) |
| Definition 10 (Compositional PMC) | `compositional_PMC` (compositional_PMC.pvs, line 44) |
| Definition 11 (Compositional PMC slot) | `slot?, slots, slotStates` (compositional_PMC.pvs, line 44) |
| Definition 12 (Partial PMC composition) | `compose_sigle_slot` (PMC_composition.pvs, line 325) |
| Definition 13 (Identifying function) | `identity_function` (compositional_PMC_sets.pvs, line 16) |
| Definition 14 (Dependency relation induced in compositional PMCs) | `<` (compositional_PMC_order.pvs, line 11) |
| Definition 15 (Minimal and maximal compositional PMCs) | `minimal?, maximal?` (compositional_PMC_order.pvs, line 14) |
| Definition 16 (Feature disabler compositional PMC) | `feature_disabler_PMC?` (PMC_composition.pvs, line 17) |
| Definition 17 (Composition factory) | `composition_factory` (compositional_reliability_models.pvs, line 21) |
| Definition 18 (Compositional probabilistic model) | `compositional_reliability_model` (compositional_reliability_models.pvs, line 90) |
| Definition 19 (Derivation by composition) | `pi` (compositional_reliability_models.pvs, line 105) |

Table C.2: Mapping between the manual and the mechanized specifications (continued)

| **Manual** | **Mechanized** |
|---|---|
| Definition 20 (Non-parametric model checking) | `alpha`<br>(DTMC.pvs, line 17) |
| Strategy 1 (Product-based analysis of annotative models) | `product_based_analysis`<br>(rome.pvs, line 21) |
| Strategy 2 (Product-based analysis of compositional models) | `product_based_analysis`<br>(rome.pvs, line 58) |
| Definition 21 (Parametric model checking) | `alpha_v`<br>(PMC.pvs, line 46) |
| Definition 22 (Expression evaluation) | `sigma`<br>(SPL_expression_evaluation.pvs, line 11) |
| Strategy 3 (Family-product-based analysis) | `family_product_based_analysis`<br>(rome.pvs, line 29) |
| Lemma 3 (Commutativity of PMC and expression evaluations) | `eval_commutativity`<br>(PMC.pvs, line 79) |
| Theorem 1 (Soundness of family-product-based analysis) | `family_product_soundness`<br>(rome.pvs, line 33) |
| Definition 23 (Expression lifting) | `eval`<br>(expression_lifting.pvs, line 20) |
| Lemma 4 (Soundness of expression lifting) | `expression_lifting_soundness`<br>(expression_lifting.pvs, line 36) |
| Definition 24 (Lifted evaluation factory) | `lifted`<br>(variability_aware_expression_evaluation.pvs, line 27) |

Table C.2: Mapping between the manual and the mechanized specifications (continued)

| Manual | Mechanized |
|---|---|
| Definition 25 (Variability-aware expression evaluation) | `sigma_v`<br>(variability_aware_expression_evaluation.pvs, line 34) |
| Theorem 2 (Soundness of variability-aware expression evaluation) | `variability_aware_expression_evaluation_soundness`<br>(variability_aware_expression_evaluation.pvs, line 37) |
| Lemma 5 (Soundness of lifted annotative evaluation factory) | `lifted_evaluation_factory_soundness`<br>(annotative_expressions_evaluation.pvs, line 22) |
| Theorem 3 (Soundness of expression evaluation using $\hat{p}$) | `lift_p_soundness`<br>(annotative_expressions_evaluation.pvs, line 27) |
| Strategy 4 (Family-based analysis) | `family_based_analysis`<br>(rome.pvs, line 39) |
| Theorem 4 (Soundness of family-based analysis) | `family_based_analysis_soundness`<br>(rome.pvs, line 44) |
| Definition 26 (Compositional evaluation factory) | `composition_evaluation_factory`<br>(compositional_reliability_models.pvs, line 62) |
| Strategy 5 (Feature-product-based analysis) | `feature_product_based_analysis`<br>(rome.pvs, line 64) |
| Theorem 5 (Soundness of feature-product-based analysis) | `feature_product_soundness`<br>(rome.pvs, line 70) |
| Definition 27 (Lifted compositional evaluation factory) | `lifted_compositional_evaluation_factory`<br>(compositional_expressions_evaluation.pvs, line 13) |
| Lemma 6 (Soundness of lifted compositional evaluation factory) | `soundness_of_lifted_compositional_evaluation_factory`<br>(compositional_expressions_evaluation.pvs, line 26) |

Table C.2: Mapping between the manual and the mechanized specifications (continued)

| **Manual** | **Mechanized** |
|---|---|
| Theorem 6 (Soundness of expression evaluation using $\varphi$) | `soundness_of_expression_evaluation_using_phi` (compositional_expressions_evaluation.pvs, line 33) |
| Strategy 6 (Feature-family-based analysis) | `feature_family_based_analysis` (rome.pvs, line 76) |
| Theorem 7 (Soundness of feature-family-based analysis) | `feature_family_soundness` (rome.pvs, line 84) |
| Lemma 7 (r-equivalence for `ITE`) | `ITE_r_equivalence` (PMC_variability_encoding.pvs, line 315) |
| Definition 28 (Variability encoding function for PMCs) | `gamma` (PMC_variability_encoding.pvs, line 333) |
| Definition 29 (Variability encoding of PMCs) | `encoded_reliability_model` (PMC_variability_encoding.pvs, line 349) |
| Theorem 8 (r-equivalence of variability encoding and derivation by composition) | `PMC_variability_encoding_soundness` (PMC_variability_encoding.pvs, line 360) |
| Definition 30 (`ITE` operator for expressions) | `ITE` (expressions_variability_encoding.pvs, line 11) |
| Lemma 8 (Extensional equality for expression `ITE`) | `expression_ITE_extensionality` (expressions_variability_encoding.pvs, line 27) |
| Definition 31 (Variability encoding function for expressions) | `gamma` (expressions_variability_encoding.pvs, line 62) |
| Definition 32 (Variability encoding of expressions) | `var_encoded_annotative_evaluation_factory` (expressions_variability_encoding.pvs, line 91) |

Table C.2: Mapping between the manual and the mechanized specifications (continued)

| Manual | Mechanized |
|---|---|
| Theorem 9 (Soundness of variability encoding for expressions) | `expressions_variability_encoding_soundness`<br>(expressions_variability_encoding.pvs, line 131) |
| Strategy 7 (Feature-family-product-based analysis) | `feature_family_product_based_analysis`<br>(rome.pvs, line 95) |
| Theorem 10 (Soundness of feature-family-product-based analysis) | `feature_family_product_soundness`<br>(rome.pvs, line 102) |
| Lemma 9 (Existence of minimal PMCs) | `minimal_exists, minimal_empty_vars`<br>(compositional_PMC_order.pvs, line 21) |
| Lemma 10 (Existence of maximal PMCs) | `maximal_exists`<br>(compositional_PMC_order.pvs, line 33) |
| Lemma 11 (Derivation by composition terminates) | `termination TCC for composition_factory`<br>(compositional_reliability_models.pvs, line 23) |
| Lemma 12 (Compositional evaluation terminates) | `termination TCC for compositional_evaluation_factory`<br>(compositional_reliability_models.pvs, line 57) |
| Definition 33 (Compositional PMC renaming) | `renaming?`<br>(PMC_renaming.pvs, line 45) |
| Definition 34 (Total PMC composition) | `compose`<br>(PMC_composition.pvs, line 542) |
| Lemma 13 (r-equivalence of total composition and evaluation) | `r_equivalence_composition_evaluation`<br>(PMC_r_equivalence.pvs, line 123) |
| Corollary 1 (r-equivalence of total composition with DTMCs and evaluation) | `r_equivalence_composition_with_DTMC`<br>(PMC_r_equivalence.pvs, line 147) |

Table C.2: Mapping between the manual and the mechanized specifications (continued)

| Manual | Mechanized |
|---|---|
| Definition 35 (ITE operator for PMCs) | ite<br>(PMC_variability_encoding.pvs, line 227) |

Table C.3: Categories of proof commands

| Category | Occurrences | % of total |
|---|---:|---:|
| Definitions | 8,600 | 31.51 |
| Quantifier | 4,527 | 16.59 |
| Type Constraints | 3,612 | 13.23 |
| Propositional | 2,872 | 10.52 |
| Lemmas | 2,532 | 9.28 |
| Decision Procedures | 2,347 | 8.60 |
| Structural | 1,425 | 5.22 |
| Equality | 584 | 2.14 |
| TCC | 372 | 1.36 |
| Extensionality | 313 | 1.15 |
| Induction | 66 | 0.24 |
| Control | 21 | 0.08 |
| Annotation | 17 | 0.06 |
| Algebraic manipulation | 3 | 0.01 |
| Rewrite Rules | 1 | 0.00 |

Table C.4: Proof commands

| Command | Occurrences | Category |
|---|---:|---|
| expand | 7,300 | Definitions |
| skeep | 3,130 | Quantifier |
| assert | 2,115 | Decision Procedures |
| typepred | 1,564 | Type Constraints |
| flatten | 1,499 | Propositional |
| expand* | 1,300 | Definitions |
| use | 1,144 | Lemmas |
| inst | 1,132 | Quantifier |
| rewrite (typepred) | 997 | Type Constraints |
| hide | 979 | Structural |
| split | 979 | Propositional |
| rewrite (sequent formula) | 946 | Type Constraints |
| rewrite (lemma) | 789 | Lemmas |
| hide-all-but | 384 | Structural |
| replace | 377 | Equality |
| subtype-tcc | 286 | TCC |
| lemma | 285 | Lemmas |
| inst (lemma) | 278 | Lemmas |
| grind | 187 | Decision Procedures |

Table C.4: Proof commands (continued)

| Command | Occurrences | Category |
|---|---:|---|
| inst-cp | 179 | Quantifier |
| lift-if | 172 | Propositional |
| replace-extensionality | 150 | Extensionality |
| case | 147 | Propositional |
| decompose-equality | 145 | Extensionality |
| case-replace | 129 | Equality |
| replace (typepred) | 105 | Type Constraints |
| skolem | 68 | Quantifier |
| beta | 66 | Equality |
| iff | 63 | Propositional |
| reveal | 59 | Structural |
| induct | 51 | Induction |
| termination-tcc | 36 | TCC |
| forward-chain | 36 | Lemmas |
| ground | 27 | Decision Procedures |
| judgement-tcc | 21 | TCC |
| postpone | 21 | Control |
| apply-extensionality | 18 | Extensionality |
| comment | 17 | Annotation |
| simplify | 12 | Decision Procedures |
| assuming-tcc | 12 | TCC |
| skosimp* | 10 | Quantifier |
| prop | 9 | Propositional |
| measure-induct+ | 8 | Induction |
| tcc | 8 | TCC |
| rule-induct | 6 | Induction |
| name | 6 | Equality |
| name-replace | 5 | Equality |
| instantiate | 4 | Quantifier |
| cond-coverage-tcc | 4 | TCC |
| case* | 3 | Propositional |
| smash | 3 | Decision Procedures |
| both-sides | 3 | Decision Procedures |
| copy | 3 | Structural |
| existence-tcc | 3 | TCC |

Table C.4: Proof commands (continued)

| Command | Occurrences | Category |
|---|---|---|
| generalize | 2 | Quantifier |
| swap-rel | 2 | Algebraic manipulation |
| cond-disjoint-tcc | 2 | TCC |
| generalize-skolem-constants | 1 | Quantifier |
| skosimp | 1 | Quantifier |
| install-rewrites | 1 | Rewrite Rules |
| replace* | 1 | Equality |
| induct-and-simplify | 1 | Induction |
| mult-ineq | 1 | Algebraic manipulation |

Table C.5: PVS specification and coverage of the original theory

| PVS Theory | Mechanized Theorems | Original Theorems | Original Definitions |
|---|---|---|---|
| PMC_composition | 200 | 0 | 2 |
| PMC_variability_encoding | 152 | 2 | 3 |
| PMC | 118 | 2 | 3 |
| PMC_renaming | 84 | 0 | 1 |
| DTMC | 81 | 0 | 1 |
| compositional_PMC | 62 | 0 | 2 |
| rational_expressions | 62 | 0 | 0 |
| PMC_r_equivalence | 59 | 2 | 0 |
| parametric_transition_matrices | 58 | 0 | 0 |
| finite_sets_aux | 39 | 0 | 0 |
| expressions_variability_encoding | 27 | 2 | 3 |
| rome | 26 | 5 | 7 |
| compositional_expressions_evaluation | 22 | 2 | 1 |
| expression_lifting | 22 | 1 | 1 |
| list_aux | 20 | 0 | 0 |
| annotative_PMC | 14 | 0 | 1 |
| annotative_expressions_evaluation | 10 | 2 | 0 |
| real_ADD | 10 | 0 | 0 |
| compositional_reliability_models | 10 | 2 | 4 |
| well_founded_lemmas | 9 | 0 | 0 |
| maybe_real | 9 | 0 | 0 |
| infinite_set_lemmas | 9 | 0 | 0 |
| compositional_PMC_order | 8 | 2 | 2 |
| compositional_reliability_models_lemmas | 6 | 0 | 0 |
| list_map_aux | 6 | 0 | 0 |
| variability_aware_expression_evaluation | 5 | 1 | 2 |
| finite_set_lemmas | 5 | 0 | 0 |
| annotative_reliability_models | 4 | 1 | 1 |
| transition_matrices_sum | 4 | 0 | 0 |
| SPL | 3 | 0 | 0 |
| compositional_PMC_sets | 3 | 0 | 1 |
| transition_matrices_map | 3 | 0 | 0 |
| compositional_factory_restrict | 2 | 0 | 0 |
| SPL_reliability | 2 | 0 | 3 |
| real_transition_matrices | 2 | 0 | 0 |
| transition_matrices | 1 | 0 | 0 |
| states | 1 | 0 | 0 |
| ADD_ops | 0 | 0 | 0 |
| ADD_def | 0 | 0 | 0 |
| SPL_expression_evaluation | 0 | 0 | 1 |

# Appendix D

# Theory Dependencies

This appendix is a compilation of dependency graphs for the main theorems presented in this work. We believe that, alongside Figure 3.7, the diagrams presented here may be useful to researchers seeking to adapt or generalize our theoretical results.

The dependency graphs are depicted in diagrams where nodes represent theory elements (i.e., theorems, lemmas and definitions), 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 D.1: Overall theory structure

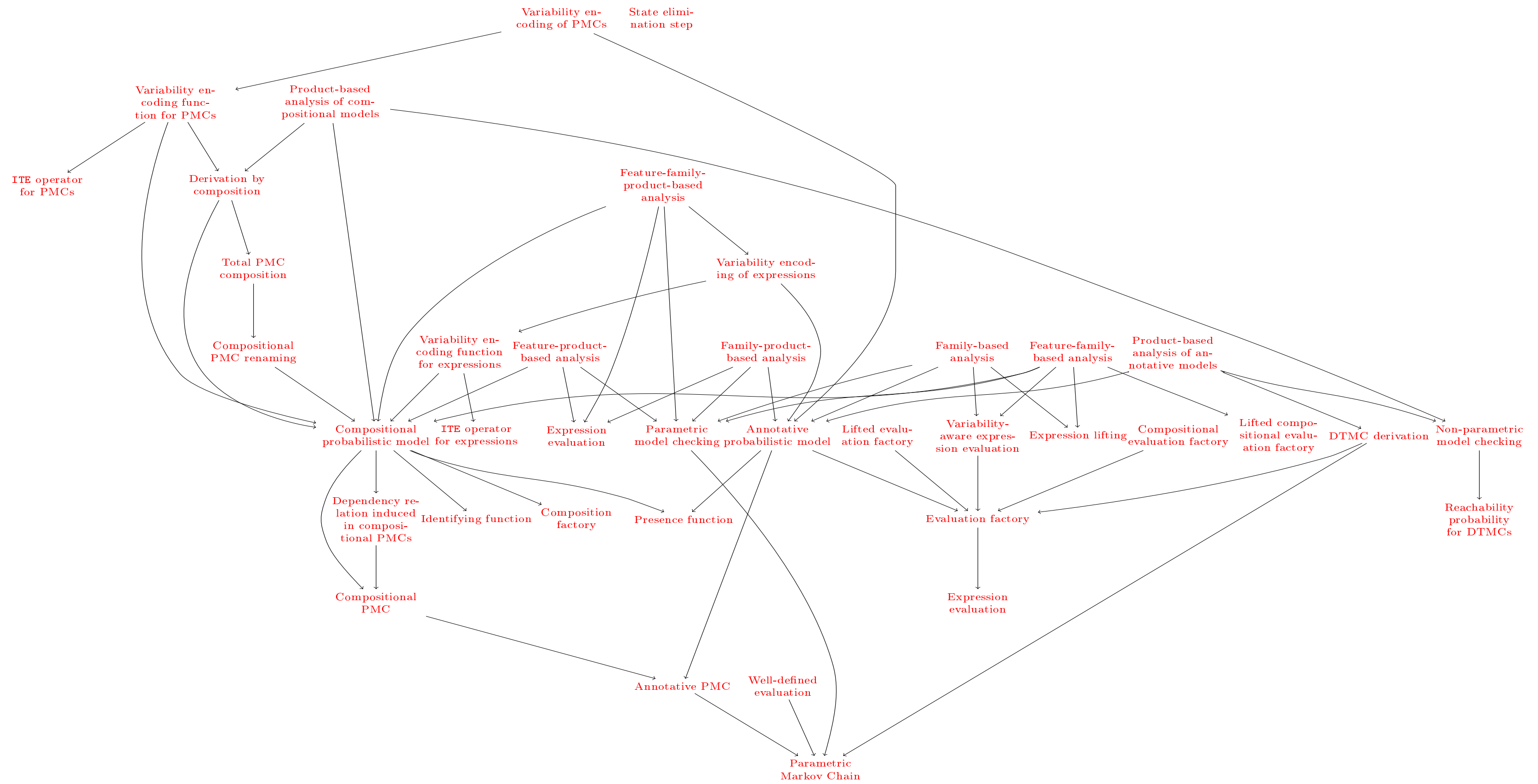



Figure D.2: Overall theory structure (only definitions)

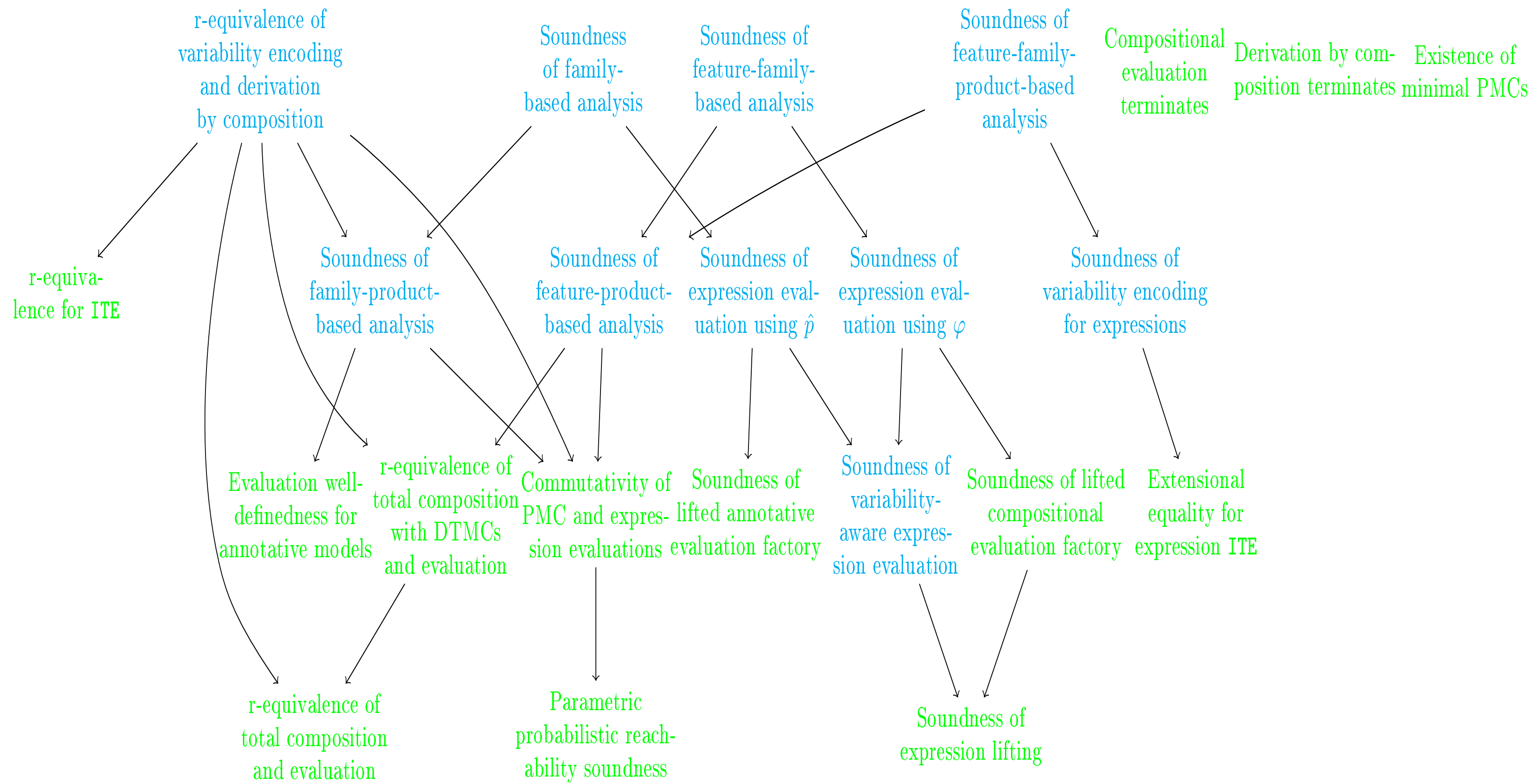



Figure D.3: Overall theory structure (only theorems and lemmas)

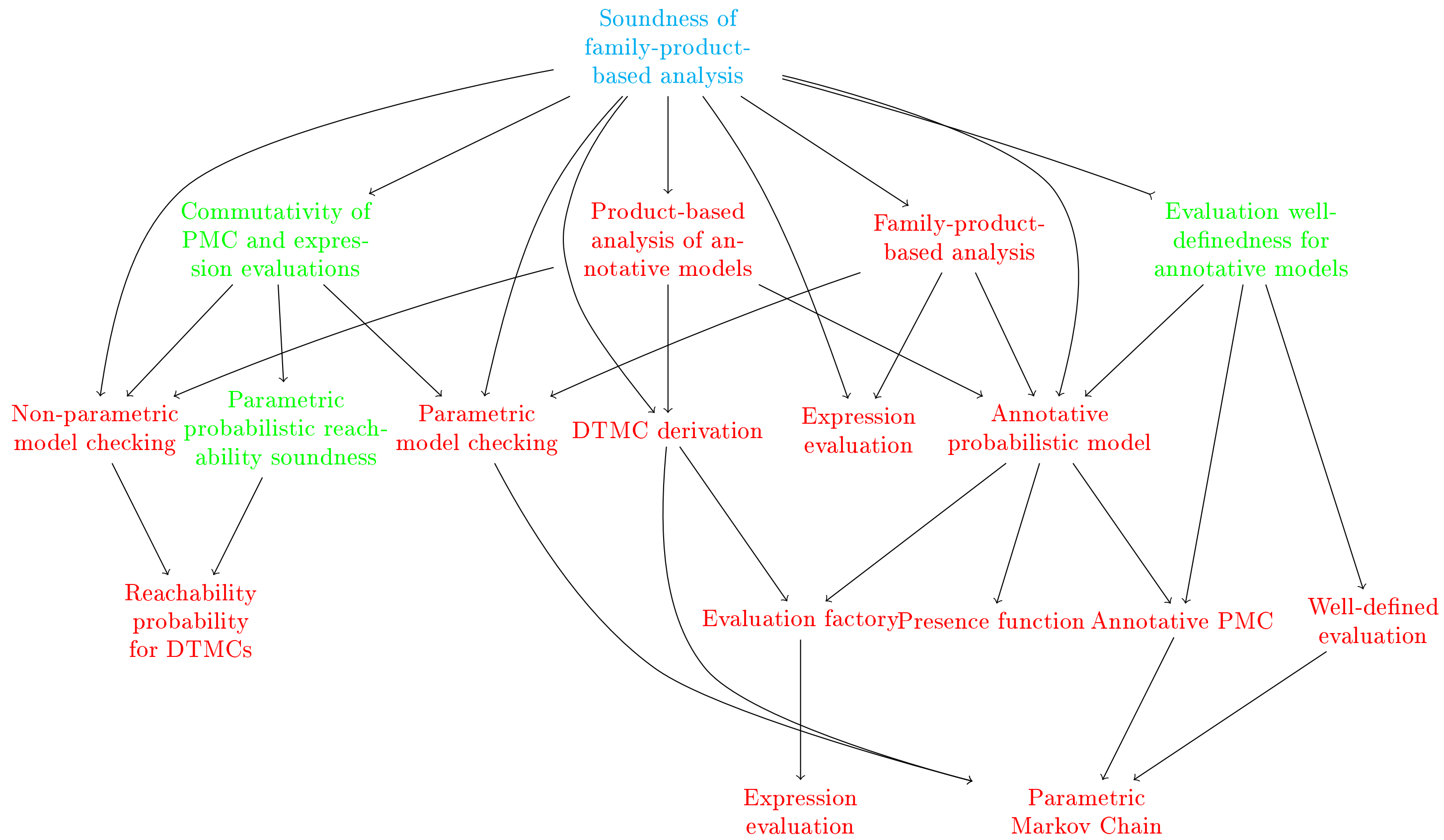



Figure D.4: Dependencies for Theorem 1 (Soundness of family-product-based analysis)

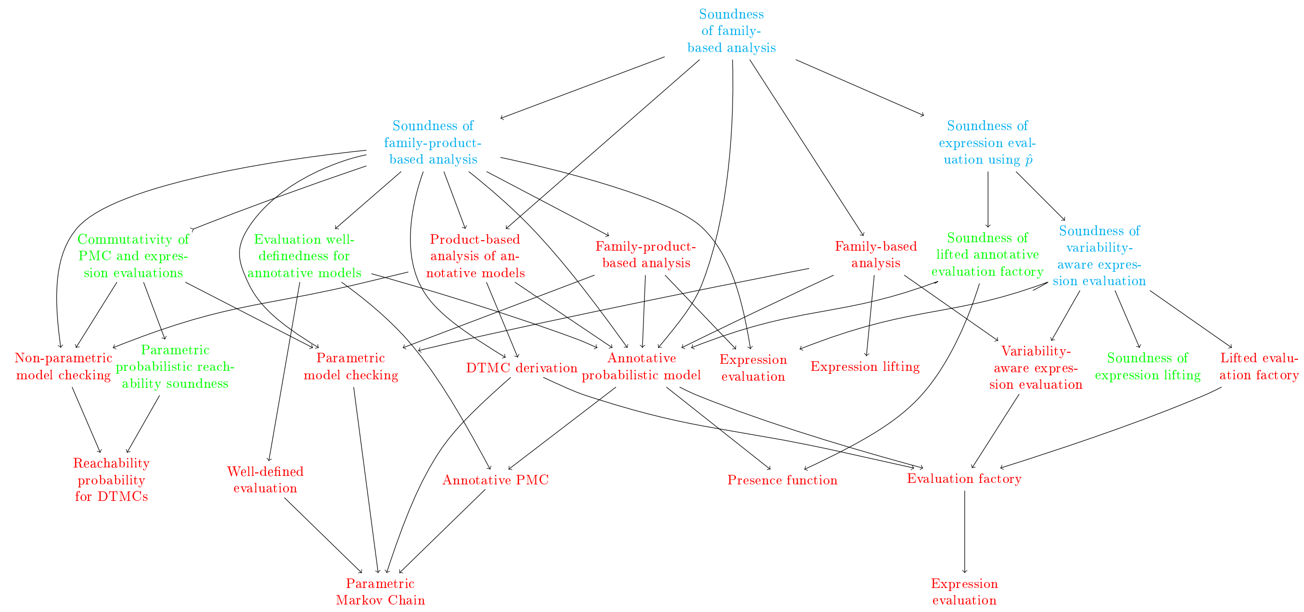



Figure D.5: Dependencies for Theorem 4 (Soundness of family-based analysis)

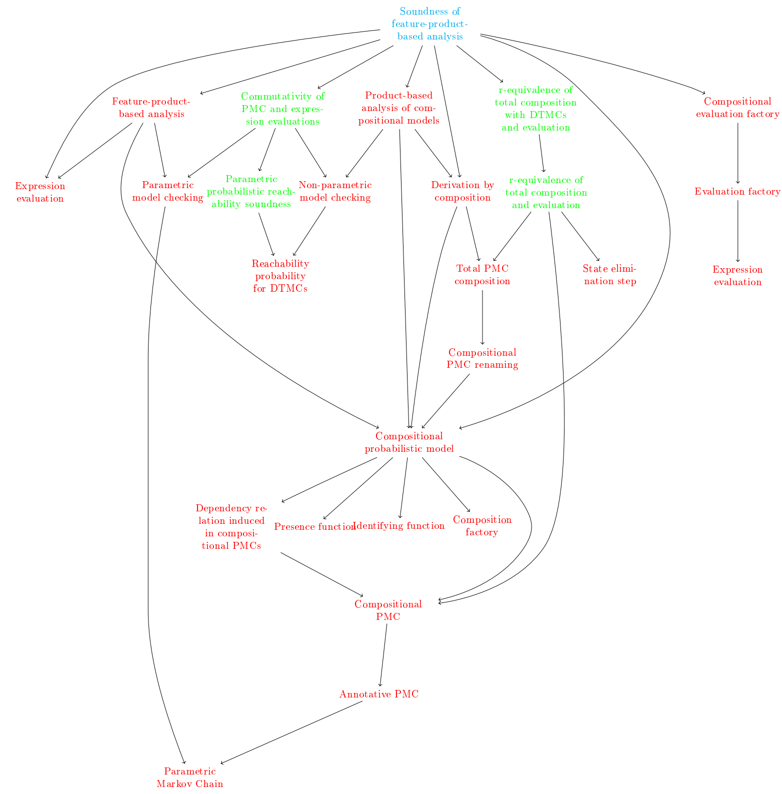



Figure D.6: Dependencies for Theorem 5 (Soundness of feature-product-based analysis)

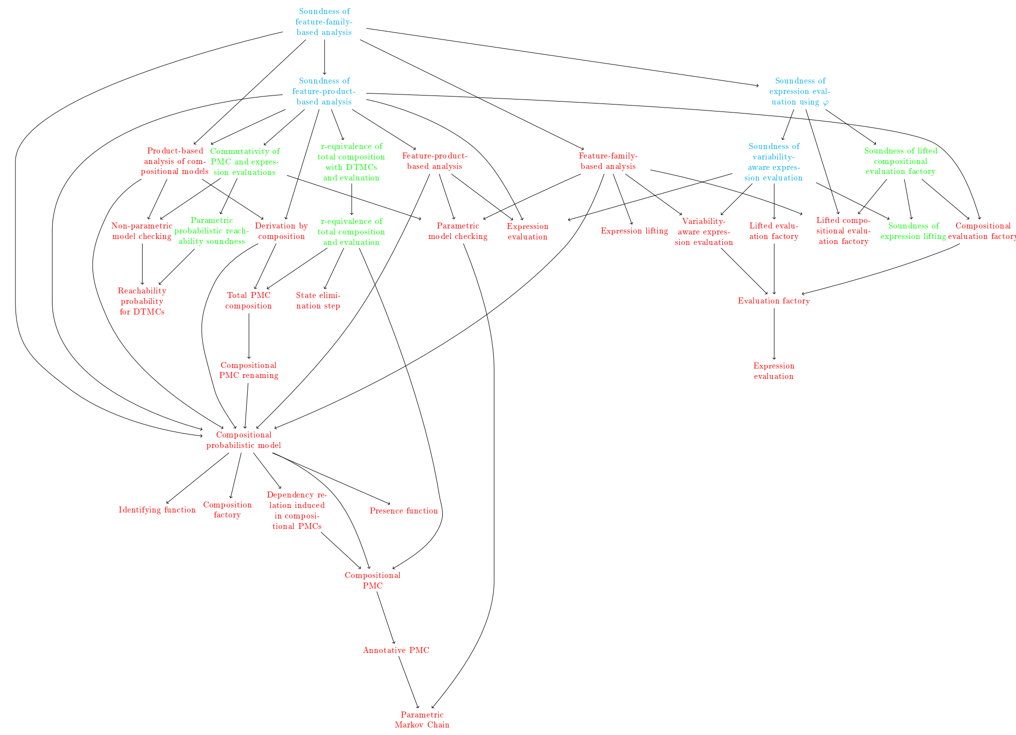



Figure D.7: Dependencies for Theorem 7 (Soundness of feature-family-based analysis)

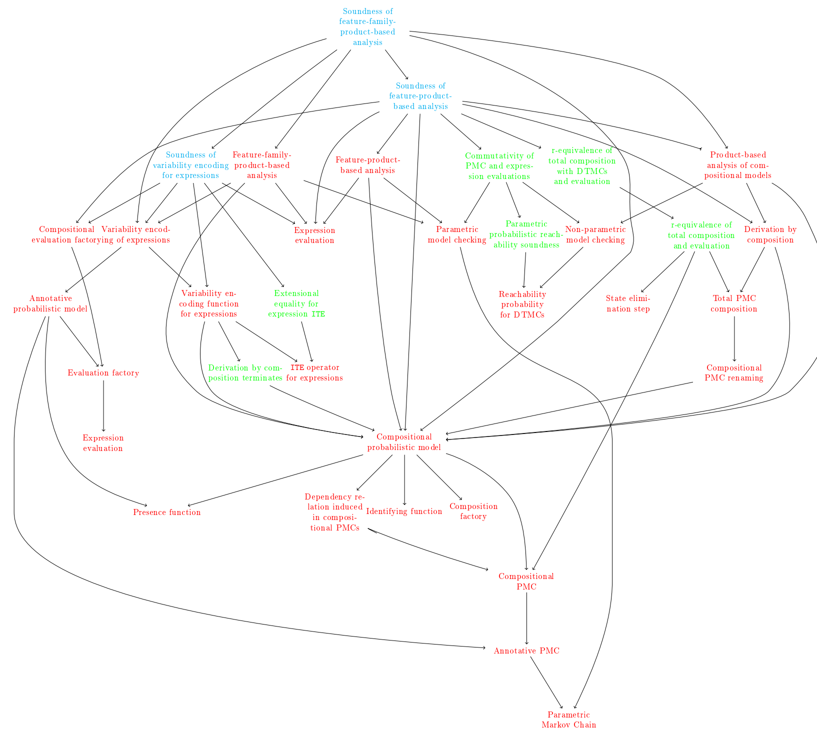



Figure D.8: Dependencies for Theorem 10 (Soundness of feature-family-product-based analysis)