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

Feature-Family-Based Reliability Analysis of Software Product Lines

André Luiz Peron Martins Lanna

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Orientador
Prof. Dr. Vander Ramos Alves

Coorientadora
Prof.a Dr.a Genaina Nunes Rodrigues

Brasília
2017

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

PS0681f

Peron Martins Lanna, Andre Luiz
Feature-Family-Based Reliability Analysis of Software Product Lines / Andre Luiz Peron Martins Lanna; orientador Vander Ramos Alves; co-orientador Genaina Nunes Rodrigues. - Brasília, 2017.
110 p.

Tese (Doutorado - Doutorado em Informática) -- Universidade de Brasília, 2017.

1. Software Product Lines. 2. Software Reliability. 3. Reliability Analysis. 4. Model Checking. 5. Software Analysis. I. Ramos Alves, Vander, orient. II. Nunes Rodrigues, Genaina, co-orient. III. Título.

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

Feature-Family-Based Reliability Analysis of Software Product Lines

André Luiz Peron Martins Lanna

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

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

Prof. Dr. Vander Ramos Alves
CIC/UnB

Prof. Dr. Rodrigo Bonifácio
CIC/UnB

Prof. Dr. Alexandre Cabral Mota
CIN/UFPE

Prof. Dr. Cecilia Mary Fischer Rubira
Unicamp

Prof. Dr. Bruno Luiggi Macchiavello Espinoza
Coordenador do Programa de Pós-graduação em Informática

Brasília, 29 de setembro de 2017

Dedicatória

À Deus e aos meus mentores espirituais,
à minha esposa Cinthia,
aos meus pais e irmãos e, por fim,
aos meus amigos que me acompanharam nessa caminhada.

Agradecimentos

À minha amada esposa e sobretudo amiga, Cinthia, que me acompanhou remota e presencialmente nas diversas fases de meu doutoramento. Muito obrigado, gatinha, pela compreensão nos momentos em que tive que me ausentar, por levantar meu ânimo e me encorajar nos momentos de desânimo, por celebrar comigo as boas notícias, enfim... Obrigado por estar sempre ao meu lado.

Aos meus pais, irmãos e cunhadas, pela confiança e pelo apoio durante todo esse tempo de estudos e trabalhos. Apesar de muitas vezes ter que abrir mão de estar com todos vocês, eu sempre soube que vocês estavam ali ao lado para o que eu precisasse e isso sempre me trouxe a confiança e o conforto necessários para seguir adiante.

Agradeço de modo muito especial ao meu orientador, Prof. Vander Alves, pelo compromisso, profissionalismo e pelas inestimáveis contribuições oferecidas ao longo de minha orientação. Obrigado por sempre me oferecer as melhores condições para realizar minha pesquisa, seja através de minha capacitação ao incentivar minha participação em valiosas escolas de versão, seja por fornecer alunos de graduação e mestrado para auxiliar em minhas tarefas. Mas, em especial, agradeço-lhe por entender os momentos difíceis pelos quais passei e que tiveram impacto em minha pesquisa. Por fim, não posso deixar de agradecer por me ensinar através da convivência o modo como um pesquisador sério conduz os seus trabalhos de pesquisa e orientação sempre buscando a investigação de temas relevantes e a excelência nos estudos. Esse ensinamento é, de fato, inestimável!

Agradeço à minha co-orientadora, Profa. Genaina Rodrigues, pelos valiosos comentários e sugestões (técnicos e pessoais) ao longo de nossa pesquisa. Por diversas vezes ela contribuiu com idéias a serem consideradas na execução do trabalho mas, também, ao dar dicas de como encarar as dificuldades típicas de um Doutorado. Aproveito para estender meus agradecimentos ao professores Pierre-Yves Schobbens e Sven Apel por disponibilizarem um tempo para reunir e contribuir de modo muito significativo com meu trabalho. Vários outros professores também contribuiram muito para meu trabalho e minha formação dentre eles os professores Azzedine Boukerche, Doron Peled e Tom Mens.

Aos colegas do PPGI, Thiago Castro, Eneias Xavier, Ricardo Chaves, Ulisses, Waldeyr, Marcos Caetano, Jeremias Gomes, Daniel Souza, Ariane, Jose e Breno. Muito obrigado a todos vocês pela companhia no dia-a-dia do PPGI discutindo sobre doutorado ou, simplesmente, conversando sobre qualquer coisa que nos pudesse descontrair. Muito obrigado, pessoal!

Por fim, agradeço de modo muito especial aos professores e amigos da Faculdade do Gama a citar: Cristiane Soares Ramos, Ricardo Ajax, Luiz Myiadaira, Rejane Figueiredo, Edna Canedo, Sergio Freitas e Gustavo Cuerva. Obrigado a todos vocês por me apoiarem nessa empreitada.

"Success consists of going from failure
to failure without loss of enthusiasm."
"Never, never, never give up."
(Winston Churchill)

Resumo

Contexto: Técnicas de verificação têm sido aplicadas para garantir que sistemas de software atinjam níveis de qualidade desejados e atenda a requisitos funcionais e não-funcionais. Entretanto a aplicação dessas técnicas de verificação em linhas de produto de software é desafiador devido à explosão combinatorial do número de produtos que uma linha de produtos pode instanciar. As técnicas atuais de verificação de linhas de produtos utilizam model checking simbólico e informações sobre variabilidade para otimizar a análise, mas ainda apresentam limitações que as tornam onerosas ou inviáveis. Em particular, as técnicas de verificação do estado da arte para análise de confiabilidade em linhas de produto são enumerativas o que dificulta a aplicabilidade das mesmas devido à explosão combinatorial do espaço de configurações.

Objetivo: Os objetivos dessa tese são os seguintes: (a) apresentar um método eficiente para calcular a confiabilidade de todas as configurações de uma linha de produtos de sotware composicional ou anotacional à partir de seus modelos comportamentais UML, (b) fornecer uma ferramenta que implemente o método proposto e, (c) relatar um estudo empírico comparando o desempenho de diferentes estratégias de análises de confiabilidade para linhas de produto de software.

Método: Esse trabalho apresenta uma nova estratégia de análise *feature-family-based* para calcular a confiabilidade de todos os produtos de uma linha de produtos de software (composicional ou anotacional). O passo *feature-based* da estratégia divide os modelos comportamentais em unidades menores para que essas possam ser analisadas mais eficientemente. O passo *family-based* realiza o cálculo de confiabilidade para todas as configurações de uma só vez ao avaliar as expressões de confiabilidade em termos de uma estrutura de dados variacional adequada.

Resultados: Os resulstados empíricos mostram que a estratégia *feature-family-based* para análise de confiabilidade supera, em termos de tempo e espaço, quatro outras estratéfias de análise do estado da arte (*product-based*, *family-based*, *feature-product-based* e *family-product-based*) para a mesma propriedade. No contexto da avaliação e em comparação com as outras estratégias, a estratégia *feature-family-based* foi a única capaz de escalar a um crescimento do espaço de configuração da ordem de 2^{20}.

Conclusões: A estratégia *feature-family-based* utiliza e se beneficia das estratégias *feature-* e *family-* ao domar o crescimento dos tamanhos dos modelos a serem analizados e por evitar a enumeração de produtos inerentes a alguns métodos de análise do estado da arte.

Palavras-chave: Linhas de produtos de software, análise de confiabilidade, verificação de modelos

Abstract

Context: Verification techniques are being applied to ensure that software systems achieve desired quality levels and fulfill functional and non-functional requirements. However, applying these techniques to software product lines is challenging, given the exponential blowup of the number of products. Current product-line verification techniques leverage symbolic model checking and variability information to optimize the analysis, but still face limitations that make them costly or infeasible. In particular, state-of-the-art verification techniques for product-line reliability analysis are enumerative which hinders their applicability, given the latent exponential blowup of the configuration space.

Objective: The objectives of this thesis are the following: (a) we present a method to efficiently compute the reliability of all configurations of a compositional or annotation-based software product line from its UML behavioral models, (b) we provide a tool that implements the proposed method, and (c) we report on an empirical study comparing the performance of different reliability analysis strategies for software product lines.

Method: We present a novel *feature-family-based* analysis strategy to compute the reliability of all products of a (compositional or annotation-based) software product line. The *feature-based* step of our strategy divides the behavioral models into smaller units that can be analyzed more efficiently. The *family-based* step performs the reliability computation for all configurations at once by evaluating reliability expressions in terms of a suitable variational data structure.

Results: Our empirical results show that our feature-family-based strategy for *reliability* analysis outperforms, in terms of time and space, four state-of-the-art strategies (product-based, family-based, feature-product-based, and family-product-based) for the same property. In the evaluation's context and in comparison with the other evaluation strategies, it is the only one that could be scaled to a 2^{20}-fold increase in the size of the configuration space.

Conclusion: Our feature-family-based strategy leverages both feature- and family-based strategies by taming the size of the models to be analyzed and by avoiding the products enumeration inherent to some state-of-the-art analysis methods.

Keywords: Software product lines, reliability analysis, model checking

Summary

1 Introduction mylinkcolor!100

1.1 Context: . . . mylinkcolor!100!b

1.2 Solution . . . mylinkcolor!100!b

1.3 Summary of Goals . . . mylinkcolor!100!b

1.4 Organization . . . mylinkcolor!100!b

2 Background mylinkcolor!100

2.1 Software Reliability . . . mylinkcolor!100!b

2.2 Reliability Analysis . . . mylinkcolor!100!b

2.2.1 Parametric Probabilistic Reachability . . . mylinkcolor!100!b

2.3 Algebraic Decision Diagrams . . . mylinkcolor!100!b

2.4 Software Product Line . . . mylinkcolor!100!b

2.4.1 Software Product Line Analysis . . . mylinkcolor!100!b

2.5 Running example . . . mylinkcolor!100!b

2.6 Conclusion . . . mylinkcolor!100!b

3 Behavioral Modeling and Reliability of Software Product Lines mylinkcolor!100

3.1 Probabilistic and Variable Behavior Modeling of Software Product Lines . . mylinkcolor!100!b

3.1.1 UML Activity Diagrams' Elements . . . mylinkcolor!100!b

3.1.2 UML Sequence Diagrams . . . mylinkcolor!100!b

3.2 Reliability of UML Behavioral Models . . . mylinkcolor!100!b

3.2.1 Reliability of software product line . . . mylinkcolor!100!b

3.2.2 Reliability of activity diagram elements . . . mylinkcolor!100!b

3.2.3 Reliability of sequence diagram models . . . mylinkcolor!100!b

3.3 Transformation from UML to FDTMC . . . mylinkcolor!100!b

3.3.1 Transformation Rules for Activity Diagram Elements . . . mylinkcolor!100!b

3.3.2 Transformation Rules for Sequence Diagram Elements . . . mylinkcolor!100!b

3.4 Reliability Equivalence of UML Behavioral Models and FDTMCs . . . mylinkcolor!100!b

3.4.1 Reliability equivalence for activity diagram . . . mylinkcolor!100!b

3.4.2 Reliability equivalence for sequence diagram . . . . mylinkcolor!100!b
3.5 Conclusion . . . . mylinkcolor!100!b

4 Feature-Family-based Reliability Analysis **mylinkcolor!100**
4.1 Transformation . . . . mylinkcolor!100!b
4.1.1 Behavioral Models . . . . mylinkcolor!100!b
4.2 Runtime dependency graph (RDG) . . . . mylinkcolor!100!b
4.3 Feature-Based analysis . . . . mylinkcolor!100!b
4.4 Family-Based Analysis . . . . mylinkcolor!100!b
4.5 Conclusion . . . . mylinkcolor!100!b

5 Proposal Evaluation **mylinkcolor!100**
5.1 Implementation . . . . mylinkcolor!100!b
5.2 Analytical Complexity . . . . mylinkcolor!100!b
5.3 Empirical Evaluation . . . . mylinkcolor!100!b
5.3.1 Subject Systems and Experiment Design . . . . mylinkcolor!100!b
5.3.2 Experiment setup . . . . mylinkcolor!100!b
5.3.3 Results and analysis . . . . mylinkcolor!100!b
5.3.4 Discussion . . . . mylinkcolor!100!b
5.4 Threats to validity . . . . mylinkcolor!100!b

6 Conclusion **mylinkcolor!100**
6.1 Future Work . . . . mylinkcolor!100!b
6.2 Related works . . . . mylinkcolor!100!b
6.2.1 Comparison to a Feature-Product-based Strategy . . . . mylinkcolor!100!b
6.2.2 Other Related Work . . . . mylinkcolor!100!b

Referências **mylinkcolor!100**

Appendix **mylinkcolor!100**

A Experiment Data **mylinkcolor!100**

B SPL Generator Tool **mylinkcolor!100**

List of Figures

2.1 Elimination of state s in the algorithm by [mycitecolor!100!black38] . . . . mylinkcolor!100!b
2.2 ADD A_f representing the Boolean function f . . . . mylinkcolor!100!b
2.3 BSN-SPL Feature Model . . . . mylinkcolor!100!b
2.4 Behavioral diagrams for BSN-SPL . . . . mylinkcolor!100!b

3.1 Initial node of a UML Activity Diagram . . . . mylinkcolor!100!b
3.2 Activity node of a UML Activity Diagram . . . . mylinkcolor!100!b
3.3 Decision node of a UML Activity Diagram . . . . mylinkcolor!100!b
3.4 Merge node of a UML Activity Diagram . . . . mylinkcolor!100!b
3.5 End node of a UML Activity Diagram . . . . mylinkcolor!100!b
3.6 Messages types of a UML sequence diagram . . . . mylinkcolor!100!b
3.7 Alternative fragment of a UML sequence diagram . . . . mylinkcolor!100!b
3.8 Loop fragment of a UML sequence diagram . . . . mylinkcolor!100!b
3.9 Optional fragment of a UML sequence diagram . . . . mylinkcolor!100!b
3.10 Associations between behavioral fragments, sequence diagrams and Optional combined fragments . . . . mylinkcolor!100!b
3.11 Transformation rule for an initial node of an activity diagram . . . . . . mylinkcolor!100!b
3.12 Transformation rule for an activity node of an activity diagram . . . . . . mylinkcolor!100!b
3.13 Transformation rule for a decision node of an activity diagram . . . . . . mylinkcolor!100!b
3.14 Transformation rule for a merge node of an activity diagram . . . . mylinkcolor!100!b
3.15 Transformation rule for an end node of an activity diagram . . . . mylinkcolor!100!b
3.16 Transformation rule for a synchrounous, asynchronous and reply messages of a sequence diagram . . . . mylinkcolor!100!b
3.17 Transformation rule for an alternative fragment of an sequence diagram . . mylinkcolor!100!b
3.18 Transformation rule for a loop fragment of a sequence diagram . . . . . . mylinkcolor!100!b
3.19 Transformation rule for an optional combined fragment of a sequence diagram mylinkcolor!100!b
3.20 Intuition of the reliability equivalence of UML and FDTMCs models . . . . mylinkcolor!100!b
3.21 Derivation tree and reliability formula computed for the activity diagram of the BSN-SPL . . . . mylinkcolor!100!b
3.22 FDTMC of the activity diagram of the BSN-SPL . . . . mylinkcolor!100!b

3.23 Derivation tree of reliability definitions for the *Sqlite* feature . . . . mylinkcolor!100!b
3.24 FDTMC of the *Sqlite* feature . . . . mylinkcolor!100!b
3.25 Derivation tree of reliability definitions for the *Oxygenation* and *Temperature* sequence diagrams . . . . mylinkcolor!100!b
3.26 FDTMC of the *Oxygenation* and *Temperature* sequence diagrams . . . . . mylinkcolor!100!b

4.1 Feature-family-based approach for efficient reliability analysis of product lines . . . . mylinkcolor!100!b
4.2 Resulting FDTMCs . . . . mylinkcolor!100!b
4.3 RDG excerpt for the BSN product line . . . . mylinkcolor!100!b
4.4 ADDs for the running example . . . . mylinkcolor!100!b

5.1 Evolution of subject systems accomplished by the SPL-Generator tool . . . mylinkcolor!100!b
5.2 Time and memory required by different analysis strategies when evaluating evolutions of Email System . . . . mylinkcolor!100!b
5.3 Time and memory required by different analysis strategies when evaluating evolutions of MinePump System . . . . mylinkcolor!100!b
5.4 Time and memory required by different analysis strategies when evaluating evolutions of BSN-SPL . . . . mylinkcolor!100!b
5.5 Time and memory required by different analysis strategies when evaluating evolutions of Lift System . . . . mylinkcolor!100!b
5.6 Time and memory required by different analysis strategies when evaluating evolutions of InterCloud System . . . . mylinkcolor!100!b
5.7 Time and memory required by different analysis strategies when evaluating evolutions of TankWar battle game . . . . mylinkcolor!100!b

List of Tables

4.1 Reliability of `rOxygenation`, `rTemperature` and `rSituation` fragments . . mylinkcolor!100!b

5.1 Initial version of product lines used for empirical evaluation . . . . mylinkcolor!100!b
5.2 Probabilistic models statistics . . . . mylinkcolor!100!b

A.1 Time in milliseconds (fastest strategy in boldface). . . . . mylinkcolor!100!b

Chapter 1

Introduction

Achieving a high quality, low costs, and a short time to market are the driving goals of software product line engineering. It aims at developing a number of software products sharing a common and managed set of features [1]. A software product line [2] is created to take advantage of the commonalities and variabilities of a specific application domain, by reusing artifacts when instantiating individual software products (a.k.a. *variants* or simply *products*). A domain variability is expressed in terms of features, which are distinguishable characteristics relevant to some stakeholder of the domain [3]. Nowadays software product line engineering is widely accepted in both industry [4, 5] and academia [6, 2, 7, 1].

Quality assurance of product lines has drawn growing attention [8, 9]. Model checking is a verification technique that systematically explores the possible states in a formal model of the system in order to find out whether a given property is satisfied or not by software[10].

Such an analysis is realized by an automated evaluation method that performs an exhaustive search over the state space that represents the software's behavior. The fulfillment of the property under investigation is formally verified in each reachable state such the model checker answers *yes* in case it is satisfied or *no*, otherwise. For the last case, the model checker also presents a counter-example to indicate how the unexpected result can be reached again [10]. The models considered and verified by model checkinng techniques are usually finite-state automata. In such models, each state represents the conditions reached by the software after each actions it takes, and such actions are associated to every transition between two states.

In special, model checking techniques are used to evaluate probabilistic properties of software. Markov Chain is a modeling notation usually employed for representing such a probabilistic behavior, which considers the probability of transitioning from a state to another depends uniquely from the current state. Discrete-Time Markov Chain (DTMC) is a kind of Markov Chain where each transition taken represents that a time unity has elapsed. 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 is useful to represent probabilities in a model whose values varies according to software or the context [11, 12].

Particularly, model checking techniques for product lines explore the space of all products in a product line by searching for execution states where functional [13, 14, 15] or non-functional [16, 11, 17, 18, 12] properties are violated [19]. Nevertheless, employing model checking techniques to verify product lines is a complex task, posing a twofold challenge [14]: (1) the number of variants whowm need to be verified may grow exponentially with the number of features, which gives rise to an *exponential blowup* of the configuration space [20, 15, 21, 6]; and (2) model checking is inherently prone to the *state-explosion problem* [10, 19] given the size (often huge) of the models to be evaluated. Therefore, model checking all products of a product line is often not feasible in practice [9].

Dependability is a non-functional software property that should be considered in a probabilistic sense and which encompasses attributes such as *availability* and *reliability* [22]. The authors stress the probabilistic nature of dependability when state "*the extent to which a system possesses the attributes of dependability should be considered in a probabilistic sense*" [22]. From the probabilistic perspective, the reliability can be defined as a probabilistic existence property [23] whereby the result of measuring the probability of reaching some desired states in a stochastic model will indicate the probability of a software successfully accomplishes its tasks.

1.1 Context:

In previous work, model checking techniques have been applied to analyze probabilistic properties of product lines, in particular, the reliability [11, 12, 18]. These approaches attenuate the complexity of analyzing probabilistic properties by exploiting, to some extent, reuse in modeling and analysis. On the one hand, non-compositional techniques exploit commonalities across products resulting into a single model representing the variability and the behavior of the product line as a whole (covering the behaviors of all products), but it may not scale due to the large state space of models generated by this modeling approach [12, 18]. On the other hand, a compositional alternative is to create and analyze isolated models for each feature and then evaluate them jointly for each configuration [11]. This approach is space-efficient, but faces the exponential blowup of the configuration space by enumerating all valid configurations, which leads to time scalability issues. In essence, both approaches have limitations in reusing analysis effort in product lines. As a result, state-of-the-art verification techniques for product-line reliability analysis are enumerative (a.k.a. *product-based*), which hinders their applicability, given the latent exponential blowup of the configuration space. Consequently, unwanted redundant computational effort is wasted on modeling and analyzing product line's models [11].

Reduce or eliminate the redundant effort when verifying SPL's models worth to be investigated due the need of scalar model checking approaches able to evaluate SPLs within time and space constraints in case such restrictions needs to be considered by the model checker. For example, Real-time systems are known by having strong time constraints to provide an answer for an event perceived by the system [24]. If such answer is to deploy a new configuration within a specific reliability threshold value, the model checker must be able to verify which SPL's products can fulfill such constraint within the deadline specified for the real-time system. A particular example is the Ambient Assisted Living (AAL) system which monitors changes at individual's health conditions in order to identify emergency conditions and performing appropriate actions [25]. According to the individual's health conditions the AAL must ensure different reliability thresholds are reached. Thus improve the scalability of reliability verification is relevant and must be investigated. The reuse of software models previously computed and evaluated seems to be a promising approach for taming the SPLs evaluation's complexity because it may decrease the verification effort, in special it may extinguish the redundant verification effort.

1.2 Solution

As the key contribution, it is presented a method to efficiently compute the reliability of all products of both compositional and annotation-based product lines, without enumerating and analyzing each of these products. In a brief, the software product line's behavior is represented by UML behavioral diagrams (namely Activity and Sequence diagrams) which, by their turn, are composed by different behavioral fragments, each one with a specific semantics and a guard condition denoting the software's context that may be observed to allow the execution of its behavior. The behavioral variability is represented for both kinds of software product lines by means of *optional* behavioral fragments with an associated propositional formula defined in terms of the features called *presence condition*. However, they differentiate by the manner how features are associated to such fragments. Meanwhile a *compositional* product line has its variability represented by distincts and well-defined behavioral modules, an *annotational* product line has the variability defined in different locations and, such variabilities points may also be nested (likewise the `ifdef` compilation's directives) [26]. The method employs a divide-and-conquer strategy in which pre-computed reliabilities of individual behavioral model fragments associated to one or more features are combined to compute the reliability of the whole product line in a single pass. Each variability point is a behavioral fragment whose guard condition denotes its presence condition by a propositional expression defined in terms of features. In a nutshell, in the first step, a *feature-based analysis* is applied to build a variable and probabilistic model per behavioral fragment and to analyze each such model using a parametric model checker. In such step, each behavioral fragment of the software product line (denoting a variability or a commonanlity) is analyzed in isolation from others fragments, where its UML representation is transformed into a probabilistic model able to represent its variable behavior. Such model is, indeed, a Discrete-Time Markov Chain (DTMC) able to represent variability. Later, in the same step, a parametric model checker is employed in order to analyze the probability of each behavioral fragment reaches its last state, which is understood as the reliability in the context of this work. Such parametrical analysis returns expressions that describe the reliability of fragments, whose parameters in a features's reliability expression represent the reliabilities of other fragments which it depends at runtime. In the second step, the method performs a *family-based* step to evaluate each expression in terms of Algebraic Decision Diagrams [27] that are used to encode the knowledge about valid feature combinations and the mapping to their corresponding reliabilities. Such step is characterized by solving all the resulting expressions from the feature-based step, taking into account the presence condition of each behavioral fragment and the well-formedness rules defined by the Feature Model. The result of the family-based evaluation of a behavioral fragment is the reliabilities values for each valid-partial configuration that satisfy both fragment's presence condition and Feature Model's rules. Such result is represented in a concise manner by means of Algebraic Decision Nodes whose structure is a tree having each level associated to a feature and each leaf node to a reliability value. Since the method is a combination of feature-based and family-based analyzes, it is effectively a *feature-family-based* analysis strategy [9], being the first of its kind for reliability analysis.

In a brief, the proposed approach differs from prior work [14, 11] in that (1) it captures the runtime feature dependencies from the UML behavioral models, (2) such dependencies are enriched with variability information extracted from the FM, (3) it computes the reliability values each feature may assume by evaluating each stochastic model considering the (partial) variability information and (4) compute the reliability of all SPL's products by explicitly reusing the reuse of each feature's evaluations.

The evaluation method is implemented in the tool ReAna (which stands for **Re**liability

Analysis), whose source code is publicly available as a free and open-source software[1]. The tool takes as input a set of UML behavioral models annotated with reliability information and the feature model of a product line, and it outputs the reliability values for the valid configurations (i.e., products) of this product line. To evaluate the time-space complexity, 120 experiments were performed to empirically compare the feature-family-based analysis strategy with the following state-of-the-art strategies [9]: *product-based*, *family-based*, *feature-product-based*, and *family-product-based*. In a brief, such strategies differentiate by the representativeness of their models and the manner how such models are traversed during the analyzis: at the one hand the feature-based method is an enumerative approach where a probabilistic model is built for each product, meanwhile at the other hand the family-based approach consists of a single model representing the whole variability of the software product line. In addition, evaluation strategies may also be comprised of two or more steps, each one performing a different analysis. The family-product-based strategy derives a behavioral model for a specific product from the model representing the behavior of whole software product line by solving its variability and then analyzing the resulting model. Other two analyses are the feature-product- and feature-family-based, which differ only by their last step. Both strategies initially perform a feature-based analysis where the behavioral fragments associated to one or more features are evaluated in isolation from the others. The second step consists of evaluating the resultant models from the first-step. In the case it is analyzed in an enumerative fashion, such an analysis is performed for all products instantiable by composing its related behavioral fragments, the second-step is performed by following a product-based strategy. Otherwise, in the case all the fragments are evaluated in a single step and results into the reliabilities of the whole configuration space, it is considered a family-based analyzis, so the whole analysis is considered a feature-family-based strategy. All these alternative strategies were implemented as variations of ReAna and used to analyze twenty variants of each of six publicly available product-line models: a system for monitoring an individual's health [12], control systems for mine pumps [28] and lifts [29], an email system [30], inter-cloud configuration [31], and a game [30]. These product lines have been used widely as benchmarks; they have configuration spaces of different sizes, ranging from dozens to billions of billions of products.

The experiment consisted of progressively increasing the number of features and the size of the behavioral models for each of the product lines, analyzing each of the evolved product lines with all analysis strategies. The results indicate that the feature-family-based strategy has the best performance in terms of time and space, being the only one that could be scaled to a 2^{20}-fold increase in the size of the configuration space for

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

reliability analysis when compared to four state-of-the-art strategies for the same property: product-based, family-based, feature-product-based, and family-product-based.

In summary, the contributions of this work are the following:

- It introduces a novel feature-family-based strategy for reliability analysis that analyzes each behavioral fragment (associated to one or more features) in isolation and combines the resulting pieces of information to compute the reliability of a given product line (Chapter 4);
- It provides a novel tool, called ReAna, implementing such feature-family-based method, to carry out the analysis of reliability of a product line from its UML behavioral diagrams and its feature model (Section 5.1);
- It reports on an empirical study comparing the performance of our feature-family-based strategy to other state-of-the-art analysis strategies, implemented as an extension of our ReAna tool (Section 5.3).

Supplementary material, including the ReAna tool and its extensions (which include all evaluation strategies considered in this work), as well as models used in the empirical evaluation and respective experimental results are publicly available for replication purposes at http://splmc.github.io/scalabilityAnalysis/.

1.3 Summary of Goals

The research has the following key goals:

- to present how the behavior of software product lines can be modeled by usual UML behavioral diagrams and evaluated following a divide-and-conquer strategy;
- to provide an evaluation method aimed for the reliability analysis of software product lines;
- to empirically compare the proposed evaluation method with other state-of-the-art evaluation strategies.

1.4 Organization

The text is organized as follows:

- Chapter 2 reviews some concepts regarding the model checking techniques and highlights its importance on the verification of software properties, reliability in special;

then some concepts about software product lines and a suitable probabilistic modeling suitable for representing its probabilistic and variable behavior. Finally, it presents the running example which will be used along the text, followed by the scope refinement of this work;

- Chapter 3 presents in details how the probabilistic and variable behavior of a software product line can be represented by means of UML behavioral diagrams and the manner how such diagrams are interrelated. Next, the notion of reliability of software product lines in the scope of UML behavioral models is presented, followed by the reliability definition of each behavioral element considered in this work. The transformation of UML behavioral diagrams into the fully probabilistic model FDTMC and an informal equivalence notion between such models (UML and FDTMC) are presented in the following.

- Chapter 4 presents the method proposed for the reliability evaluation of software product lines. Initially, the transformation step to create FDTMCs from UML models by applying the transformation rules described in Chapter 3 is presented, followed by the data structure created to jointly represent the behavioral and probabilistic information of the software product line. In the following, it is presented the analysis of the probabilistic models by the *feature-based* step and the manner how the reliability of the whole software product line is computed in a single pass by the *family-based* step. Finally, the tool support that implements the method is presented in the following.

- Chapter 5 initially presents how the evaluation method was implemented by an open and publicly available tool. Then two evaluation methods are described such the first one is an analytical evaluation of the method in contrast to the related work most similar for the evaluation of probabilistic models of software product lines. The other evaluation is an empirical evaluation that compares the evaluation method hereby presented with other 4 state-of-the-art evaluation strategies.

- Chapter 6 presents the final remarks, the comparisons with related works and list the topics to be investigated in the future.

Chapter 2

Background

This chapter provides an overview of fundamental concepts related to the work and a running example to guide the presentation of the evaluation method in later sections.

2.1 Software Reliability

Probabilistic verification techniques have been used in the past to substitute the concept of absolute correctness by bounds on the probability that certain behavior may occur [23]. Based on probabilistic models, it is possible to specify probabilistic system behavior due to, e.g., intrinsically unreliable hardware components and environmental characteristics. Reliability can be defined as a probabilistic existence property [23], in the sense that it is given by the probability of eventually reaching some set of *success* states in a probabilistic behavioral model of a system.

This means the reliability of a system is the probability that, starting from an initial state, the system reaches a set of *target* (also *success*) states. This value is called *reachability probability*. To analyze this property, we first model the system's behavior as a DTMC—a tuple $(S, s_0, \mathbf{P}, T)$, where S is a set of states, $s_0 \in S$ is the initial state, $\mathbf{P}$ is the transition probability matrix $\mathbf{P} : S \times S \to [0, 1]$, and $T \subseteq S$ is the set of target states. 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. 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.

The reachability probability for a DTMC can be computed using probabilistic model checking algorithms, implemented by off-the-shelf tools [10, 32]. 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 [10], 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:

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

Whenever T is a singleton $\{t\}$, we write $Pr^{\mathcal{D}}(s, t)$ to denote $Pr^{\mathcal{D}}(s, T)$.

In a product line, different products give rise to distinct behavioral models. To handle the behavioral variability that is inherent to product lines, we resort to *Parametric Markov Chains* [33].

2.2 Reliability Analysis

Reliability analysis can be defined as a probabilistic existence property [34]. This means the reliability of a system is the probability that, starting from an initial state, the system reaches a set of *target* (also *success*) states. This value is called *reachability probability*. To analyze this property, we first model the system's behavior as a DTMC—a tuple $(S, s_0, \mathbf{P}, T)$, where S is a set of states, $s_0 \in S$ is the initial state, $\mathbf{P}$ is the transition probability matrix $\mathbf{P} : S \times S \to [0, 1]$, and $T \subseteq S$ is the set of target states.[1] 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

[1] This definition departs from the one by Baier and Katoen [10] 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).

and edges link states to their successors. 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.

The reachability probability for a DTMC can be computed using probabilistic model checking algorithms, implemented by off-the-shelf tools [10, 32]. 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 [10], that suits the purpose of this work.

Property 2 (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:

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

Whenever T is a singleton $\{t\}$, we write $Pr^{\mathcal{D}}(s,t)$ to denote $Pr^{\mathcal{D}}(s,T)$.

In a product line, different products give rise to distinct behavioral models. To handle the behavioral variability that is inherent to product lines, we resort to *Parametric Markov Chains* [33].

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 [35, 36, 37].

Definition 1 [Parametric Markov Chain] A Parametric Markov Chain [38] is defined as a tuple $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$, where S is a set of states, s_0 is the initial state, $X = \{x_1, \ldots, x_n\}$ is a finite set of parameters, $\mathbf{P}$ is the transition probability matrix $\mathbf{P} : S \times S \to \mathcal{F}_X$, and $T \subseteq S$ is the set of *target* (or *success*) states. The set $\mathcal{F}_X$ comprises the *rational expressions* over $\mathbb{R}$ with variables in X, that is, fractions of polynomials with Real coefficients. This way, the semantics of a rational expression ε is a *rational function* $f_\varepsilon(x_1, \ldots, x_n) = \frac{p_1(x_1,\ldots,x_n)}{p_2(x_1,\ldots,x_n)}$ from $\mathbb{R}^n$ to $\mathbb{R}$, where p_1 and p_2 are

Real polynomials. For brevity, we hereafter refer to rational expressions simply as *expressions*.

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

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 ε, then $\varepsilon[X/u] = \varepsilon[(X \cap X')/u]$.

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 2 [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)

[2] Hahn et. al. [38] actually define it in a more general way as a partial function. However, for our purpose, it suffices to consider total functions.

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

2.2.1 Parametric Probabilistic Reachability

To compute the reachability probability in a model with variable transitions, we use a parametric probabilistic reachability algorithm. A parametric model checking algorithm for probabilistic reachability takes a PMC $\mathcal{P}$ as input and outputs a corresponding expression ε representing the probability of reaching its set T of target states. Hahn et. al [38] present such an algorithm and prove that evaluating ε with an evaluation u yields the reachability probability for the DTMC induced in $\mathcal{P}$ by the same evaluation u.

Figure 2.1 [38] illustrates a single step of this parametric probabilistic reachability algorithm. The main idea is that, for a given state s, the probability of one of its predecessors (s_1) reaching one of its successors (s_2) is given by the sum of the probability of transitioning through s and the probability of bypassing it. 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_2. Hence, the sum of probabilities for these paths correspond to the infinite sum $\sum_{i=0}^{\infty} p_a(p_c)^i p_b = p_a(\sum_{i=0}^{\infty} p_c^i)p_b = p_a \frac{1}{1-p_c} p_b$.[3]

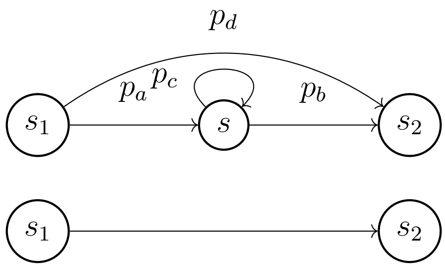
Definition 3 [State elimination step] Given a PMC $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$ and an arbitrary state $s \in S$, a state elimination step of the algorithm by [38] updates the transition matrix $\mathbf{P}$ to $\mathbf{P}'$, such that, for all states $s_1, s_2 \in S \setminus \{s\}$,

$$\mathbf{P}'(s_1, s_2) = \mathbf{P}(s_1, s_2) + \mathbf{P}(s_1, s) \cdot \frac{1}{1 - \mathbf{P}(s, s)} \cdot \mathbf{P}(s, s_2)$$

The soundness of the parametric probabilistic reachability algorithm [38] is expressed by the following lemma.

Lemma 1 (Parametric probabilistic reachability soundness)**.** Let $\mathcal{P} = (S, s_0, X, \mathbf{P}, T)$ be a PMC, u be a well-defined evaluation for $\mathcal{P}$, and ε be the output of the parametric probabilistic reachability algorithm by Hahn et. al [38] for $\mathcal{P}$ and T. Then, $Pr^{\mathcal{P}_u}(s_0, T) = \varepsilon[X/u]$.

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



$$p_a \frac{1}{1-p_c} p_b + p_d$$

Figure 2.1: Elimination of state s in the algorithm by [38]

Demonstração. The elimination algorithm [38] 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. [38] for more details on the algorithm itself and the proof mechanics. □

2.3 Algebraic Decision Diagrams

An Algebraic Decision Diagram (ADD) [39] is a data structure that encodes k-ary Boolean functions $\mathbb{B}^k \to \mathbb{R}$. As an example, Figure 2.2 depicts an ADD representing a binary function f.

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

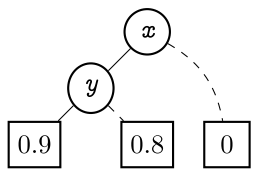


Figure 2.2: ADD A_f representing the Boolean function f

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

In the example, to evaluate $f(1, 0)$, we start in the `x` node, take the solid edge to node `y` (since the actual parameter x is 1), then take the dashed edge to the terminal 0.8. Thus, $f(1, 0) = 0.8$. Henceforth, we will use a function application notation for ADDs, meaning that, if A is an ADD that encodes function f, then $A(b_1, \ldots, b_k)$ denotes $f(b_1, \ldots, b_k)$.

For brevity, we also denote indexed parameters $b_1, \ldots, b_k$ as $\bar{b}$, and the application $A(\bar{b})$ by $[\![A]\!]_{\bar{b}}$.

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 [39]. 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. 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

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

More details on the algorithms for ADD operations are outside the scope of this work and can be found elsewhere [39].

2.4 Software Product Line

Software product lines have gained momentum in the software industry as they provide a mass customization by building individual products (tailored for the customer's requirements) from a set of reusable parts. Since a software product line is defined aiming to reuse software parts to build a product, its usage brings the benefits of improved quality of the products it creates meanwhile it reduces the development costs and time to market.

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 [2].

The main element in the software product-line engineering is to manage the variability which means having the ability to change or customize a system [5]. Such a variability is usually expressed in terms of features which are used to specify and communicate commonalities and differences of the products the software product line can instantiate. They are represented graphically by a *feature diagram*, which is a feature model [3, 40] depicted as a tree that captures the existing dependencies and constraints among the features [6]. A product of a product line is specified by a valid feature selection that fulfills all feature dependencies.

A software product line can be built using an annotation-based in case each feature is marked accordingly in a common code base, such a product is formed by removing codes related to features that do not comprise the product. Also, a software product lin can be build by a compositional approach, where each feature is implemented in a distinct unit and a product is build by composing the elements regarding its feature selection [26].

In addition, each product instantiated from a software product line has different attributes and characteristics which can be subject to some kind of analysis, in special its non-functional requirements. Some analysis techniques usually employed to the analysis of usual software are being adapted to analyze the products a software product line may instantiate as, for example, type checking, model checking, static analysis and theorem proving [9]. In particular, such task may not be feasible in practice [14] since the number of products a software product line may instantiate can be huge, sometimes it is exponential to the number of features (indeed a software product line may, in the worst case, instantiate $2^{|F|}$ products, where $|F|$ denotes the number of features.

2.4.1 Software Product Line Analysis

To analyze the behavior of a product line, it is useful to embed its inherent variability in such a probabilistic model. A possible approach is to use *parametric DTMCs* (PDTMC) [41], which augment DTMCs with transition probabilities that can be expressed as variables. A PDTMC is a DTMC whose probability matrix takes values from a set X of strictly positive parameters. A PDTMC gives rise to a family of DTMCs by instantiating the formal parameters to values with an instantiation function $\kappa : \mathbb{Q}_+ \cup X \mapsto [0, 1]$. For a parametric DTMC D_X and an instantiation function $\kappa, \kappa(D_x)$ denotes the DTMC whose probability matrix is given by instantiating D_X's formal parameters. For PDTMCs, the reliability analysis problem can be solved by a *parametric reachability* algorithm [38], which outputs a rational expression (a fraction of two polynomials) on the same variables as the ones in the input parametric model. The idea behind this technique is that evaluating the variables in the rational expression yields the reliability value of the DTMC that would be obtained by an equivalent evaluation of the variables in the PDTMC. However, this behavioral representation does not take a variability model (e.g., a feature model) into account, and thus is not sufficient for representing *possible* behavior in a product line (i.e, behavior of actual products).

Several analysis techniques have been proposed by researchers for software product lines, each one taking a particular property into account. To help researchers and practitioners understand the similarities and differences among such techniques, Thüm et. al [9] propose a classification of the existing techniques, which is followed in this work. In this context, a *product-based* reliability analysis operates only on derived (non-variable) UML behavioral models, whereas the variability model may be used to generate the models. As it is a brute-force strategy, it is only feasible for product lines with few products. In contrast, the *family-based* strategy for reliability analysis operates over variant-rich UML behavioral models and incorporates the knowledge about valid feature combinations. In a *feature-based* analysis strategy, the reliability of UML behavioral models related to each individual feature is analyzed in isolation from the others, i.e., interactions among features and the knowledge about valid feature combinations are not incorporated into the analysis.

Other evaluation strategies may be formed by combining two or more strategies aforementioned [9]. For instance, a *feature-product* analysis consists of a feature-based analysis step followed by a product-based analysis, such that the result of the feature-based analysis is reused by the product-based analysis. In the context of reliability, the reliability of UML behavioral models related to each feature is first evaluated in isolation and then the analysis result is reused when enumerating and evaluating the reliability of each non-variant UML behavioral model of the product line. In the opposite, the *feature-family based* consists of evaluating each feature in isolation (ie. a feature-based step) followed by the family-based evaluation step when each features evaluation's results are reused jointly with the knowledge about all valid configurations. Both evaluation strategies follows a compositional strategy to face the scalability issues. The compositional analysis allows to evaluate models' fragments in isolation from the others and compose such partial results in a latter step, what diminishes the evaluation effort in comparison to non-compositional analysis [42].

Although other combined evaluation strategies are possible, the aforementioned strategies suffice as contrast to the hereby proposed strategy. For more information regarding the remaining strategies, please refer to [9].

Featured Discrete-time Markov Chains (FDTMC) [12] are probabilistic models that properly handle product-line variability. They can be thought as DTMCs that, instead of transition probabilities, have transition *probability profiles*. These profiles are functions $[\![FM]\!] \rightarrow [0,1]$ that map a configuration to a probability value, where $[\![FM]\!]$ denotes the set of valid configurations of the feature model FM. Rodrigues et. al. [12] proposed a method to encode an FDTMC as a PDTMC, enabling its analysis by off-the-shelf parametric model checkers. The present work leverages the view of Rodrigues et. al. [12] of FDTMCs as PDTMCs for the purpose of compositional reliability analysis.

2.5 Running example

To illustrate the concepts presented throughout this thesis, it will be considered an example of a simple product line within the medical domain, for which reliability is considered the major requirement [43]: the Body Sensor Network (BSN) product line is a network of connected sensors that capture vital signs from an individual and send them to a central system to analyze the collected data and identify critical health situations [12]. This product line has software components that interpret data provided by the sensors and analyze an individual's health situation, as well as components for data persistence in a database or memory. The set of possible configurations for this product line is defined by its feature model (Figure 2.3), in which wireless sensors are grouped by feature *Sensor*, software components for interpreting health information are grouped by feature *SensorInformation*, and the alternatives for data persistence are grouped by feature *Storage*.

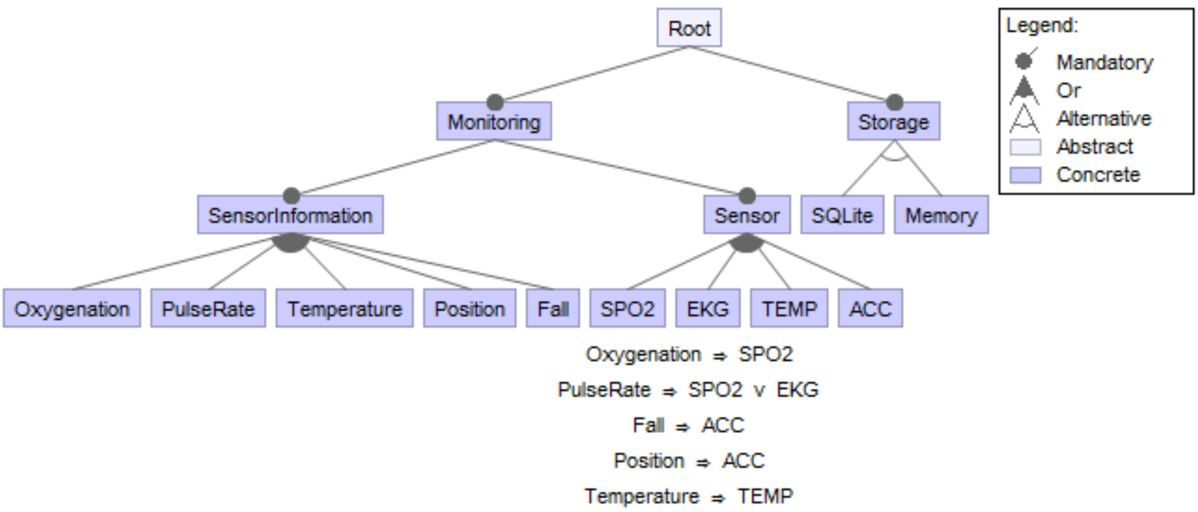
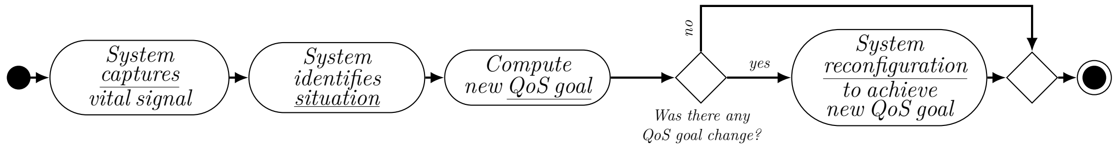


Figure 2.3: BSN-SPL Feature Model

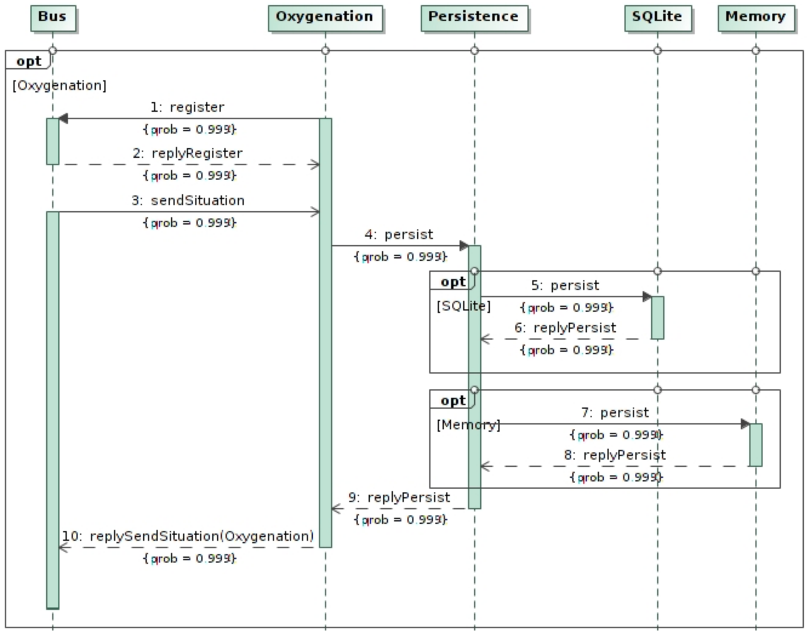
To continuously monitor an individual's health situation, the BSN product line has a control loop comprised of four activities: capture data coming from sensors, process information about the health condition, identify health goal changes, and reconfigure the system if necessary. This control loop represents the coarse-grained behavior of the BSN product line and it is modeled by the activity diagram shown in Figure 2.4a, with each activity being represented in detail by an associated sequence diagram involving the software components and their behavior. The underlined words in the activities nodes are the terms by each activity will be referred along this text. Every product instantiated from the BSN product line executes this control loop and, whenever the individual's health condition changes and this triggers a quality-of-service (QoS) goal change, another product is instantiated from this product line with the desired behavior to reach the desired QoS goal. By its turn, the behavioral representation provided by sequence diagrams is considered fine-grained since its elements are able to represent the software components enrolled in a task execution, the manner how the interactions between such components happens, in addition to behavioral branches, loops and variability. In special, sequence diagrams play the role of representing the behavioral variability due the software product line where necessary by means of guard conditions involving the presence of features (a.k.a *presence conditions* [44]).

For instance, Figures 2.4b and 2.4c present an excerpt of the sequence diagram associated with the activity *System identifies situation* (Figure 2.4a). This activity consists of processing and persisting data regarding the individual's health condition, in particular sensor information, represented by feature *SensorInformation* and its child features in Figure 2.3. Figure 2.4b depicts the behavior associated with the computation and persistence of the individual's oxygenation. Such a behavior is defined by the messages exchanged between five software components, whose roles are data processing (*Oxygenation*) and persistence (*Persistence*, *SQLite* and *Memory*—*Persistence* dispatches calls to the concrete persistence engines), and components for communication and coordination (*Bus*). Each message is named according to its task and has an associated probability value `prob` to represent the reliability of the communication channel between the components comprising the interaction. The reliability is given by the product of (a) the probability that the required message arrives at the receiver component and (b) the receiver component's reliability (i.e., the probability that it performs the required task without failure). For the BSN product line, we assume that all channels have the minimal reliability 0.999. The same understanding described above applies to the sequence diagram of Figure 2.4c since it processes and persists data regarding the individual's temperature.

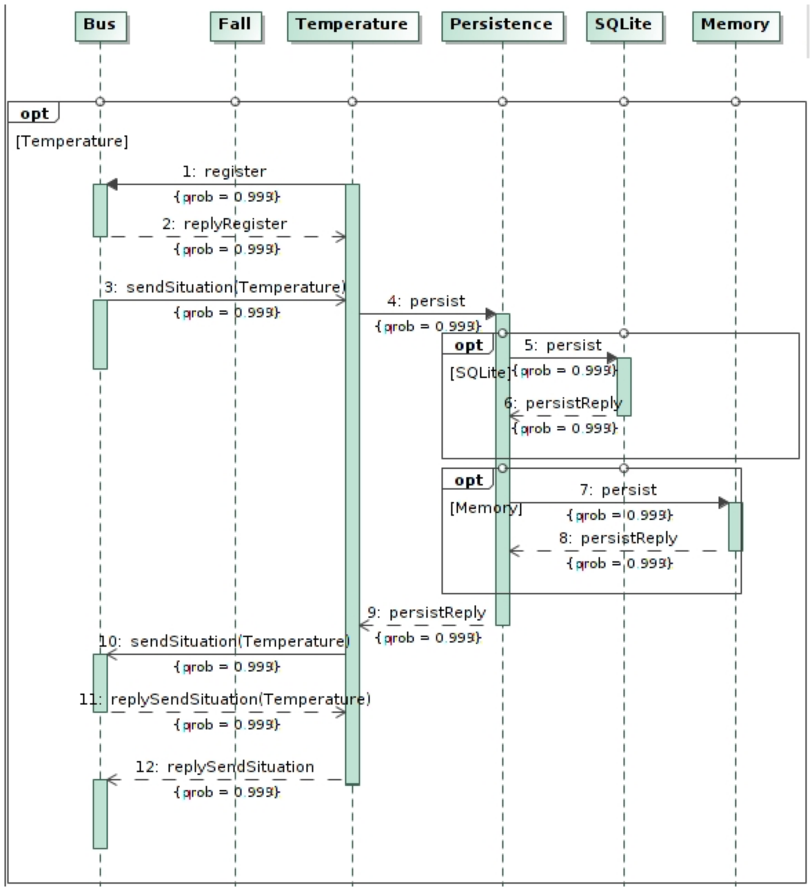
The guard condition at the top level of the sequence diagram presented in Figure 2.4b is the atomic proposition `Oxygenation`. This means that the enclosed behavior is associated with the presence of the *Oxygenation* feature in a given configuration. This behavior, in turn, has two variants, according to the chosen mechanism for data persistence. The



(a) Activity Diagram representing the control loop of BSN-SPL



(b) Sequence diagram (excerpt) associated with the activity *system identifies situation*, for processing and persisting Oxygenation information.



(c) Sequence diagram (excerpt) associated with the activity *system identifies situation*, for processing and persisting Temperature information.

Figure 2.4: Behavioral diagrams for BSN-SPL

optional fragment whose guard condition is `SQLite` models the behavior of persisting data in a database whenever feature *SQLite* is part of a configuration. Likewise, the optional fragment associated to the presence of the feature *Memory* (i.e., the fragment with the `Memory` guard) models persistence on secondary memory. In an analogous way the sequence diagram represented in Figure 2.4c is associated to the presence of *Oxygenation* feature, since its presence condition is defined by the atom `Oxygenation`. It also has two variability points for data persistence related to *SQLite* and *Memory* features, respectively.

Note that the dynamic behavior of the BSN product line does not affect the method to reliability analysis, since it only considers the execution of tasks up to reconfiguration (Figure 2.4a). Moreover, the approach is entirely based on design-time artifacts. For a deeper discussion of how the BSN product line is engineered for reconfiguration and of how the reliability computation affects this dynamic behavior, please refer to [45]

2.6 Conclusion

This chapter presented the main topics related to the verification of probabilistic properties of software product line. The verification of such properties is indeed important to ensure the desired quality level of the products a software product line. For such verification the model checking techniques play a major role but it faces challenges regarding the sizes of the configuration space and the evaluated models. Nowadays evaluation techniques propose some improvements in order to reduce the evaluation effort, like the use of symbolic model checkers, but there is still space for improvements.

The work hereby presented is, thus, aimed to explore the verification of probabilistic properties in the context of software product lines, in special, the reliability property. Such property can be evaluated on probabilistic models as the reachability measure of states considered successfull. For such evaluation, the proposed evaluation method seeks to tame the required effort by employing a *feature-family* strategy that divides the behavioral models into smaller models, analyze each one in isolation and later reuse their results to compute the reliability of the whole software product line.

Chapter 3

Behavioral Modeling and Reliability of Software Product Lines

To evaluate probabilistic properties of a software product line, reliability in particular, initially it is necessary representing the variable behavior jointly with the probabilistic information. Briefly, such an information represents the success and failure probabilities of executing the communications between software components. Both behavioral variability and probabilistic information of software product lines can be represented by the UML activity and sequence diagrams. Later, such diagrams can be transformed into their respective fully probabilistic models (FDTMCs), which must represent the states variation of the context comprising all products of the software product line.

The software product line's behavior can be considered at two abstraction levels. The high level is a coarse-grained representation that employs the UML activity diagram for modeling the set of activities executed by *all* products. The low level is a fine-grained representation whose role is to model the whole variable and probabilistic behavior of a software product line. Since the variable behavior is defined by the interaction among software components, such behavior is modeled by means of UML sequence diagrams. To represent the probabilistic information of the behavior represented by both activity and sequence diagrams, their semantics can be extended by the UML MARTE [46] profile. Thus, the joint representation of behavioral variability and probabilistic information in UML behavioral diagrams is the suitable notation for modeling the probabilistic behavior of a software product line.

The evaluation of software's probabilistic property consists of analyzing whether a property specification is fulfilled in a probabilistic model. In the case of software product lines such a probabilistic model must also address the inherent behavioral variability and the Feature Discrete-Time Markov Chain (FDTMC) is a suitable modeling notation. As previously mentioned, an FDTMC is a Discrete-Time Markov Chain (DTMC) endowed with variability for representing all products' behavior (c.f. Section 2.4.1), while the reliability property is defined as the reachability measure that expresses the probability of reaching a set of sucessfull states on a probabilistic model [23].

This chapter presents how the variable and probabilistic behavior of a software product line can be modeled by UML behavioral diagrams (activity and sequence diagrams) and later transformed into FDTMCs. The behavioral modeling of software product lines is addressed in Section 3.1 that introduces the coarse-grained behavioral representation by UML activity diagrams (Section 3.1.1), followed by the probabilistic and variable behavioral representation provided by UML sequence diagrams (Section 3.1.2). Section 3.2 introduces the reliability notion using DTMCs for UML behavioral diagrams and how it is considered in the context of software product lines. Section 3.3 presents a set of transformation rules for creating FDTMC models from UML behavioral models. Section 3.4 demonstrates evidences that the reliability computed based on UML behavioral diagrams and the reliability computed based on its corresponding FDTMCs are equivalent, which supports the correctness of transformation rules. Finally, Section 3.5 presents concluding remarks.

3.1 Probabilistic and Variable Behavior Modeling of Software Product Lines

Representing the software's characteristics by models is useful to preview and to analyze its diverse properties and behavior. Among the notations for software representation the Unified Modeling Language (UML) stands out as it provides manifold diagrams to address the different software's characteristics. Within the range of UML diagrams the activity diagram is a high level and coarse-grained behavioral representation that is usually employed to represent the software's main tasks and their execution order. The sequence diagram is a fine-grained behavioral representation that details how software components interact during a task execution. The UML MARTE profile augments the semantics of activity and sequence diagrams by associating probabilistic information to their behavioral elements.

The representation of the software product line's behavior resembles the behavioral representation of an usual software. The differences arise because the behavioral variability inherent to software product lines must be addressed by the same UML behavioral elements employed at ordinary software's models. In addition, such behavioral representations must express the probabilistic information of all products a software product line can instantiate. In the following, each UML behavioral element and its associated probabilistic information will be presented in the context of software product lines modeling.

3.1.1 UML Activity Diagrams' Elements

The coarse-grained behavioral model of a software product line is represented by a UML activity diagram enriched with probabilistic information in order to represent which are the main software product line's activities and how they are arranged and executed by *all* products. In this representation level, common flows are the tasks sequences that all products execute, which is not referred to the software components interactions shared by all products. The elements considered for such modeling level are the *Initial*, *Activity*, *Decision*, *Merge* and *Final* nodes. Each element and its meaning in the context of software product line's behavioral modeling is described in the following. Additional constraints are represented next to the element in a gray box.

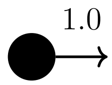


Figure 3.1: Initial node of a UML Activity Diagram

Initial node: the initial node is represented only once in a UML activity diagram by the filled circle shown by Figure 3.1. It is the execution starting point of an activity diagram and it has only one direct successor element. Since the initial node does not have any associated interaction between software components, it has no failure chances so its execution flows directly to its immediate successor, that is represented by the outgoing edge having 1.0 as probability value.

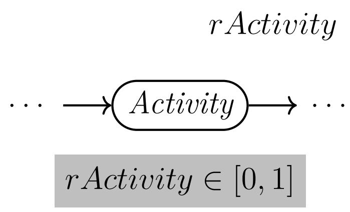


Figure 3.2: Activity node of a UML Activity Diagram

Activity: the activity node is represented by the named rounded rectangle shown by Figure 3.2 and it is responsible to represent a stage of the whole behavior modelled by the activity diagram. It has an incoming and outgoing edges to denote when its execution starts and when finishes, respectively. Each activity comprises a set of communications among several software components such the probability value of the outgoing edge represents the probability which its associated behavior is executed without errors occurrences.

Such a probability value is given by computing the reliability of its associated UML sequence diagram. Since the activity's reliability depends on the computed reliability of its associated sequence diagram and such diagram adresses the behavioral variability of the software product line, the outgoing edge's probability is represented by a variable defined in $[0, 1]$. By convention, such variable is named as the activity name with the 'r' prefix standing for "reliability".

Considering the running example of Section 2.5 each activity node represented by Figure 2.4a has its behavior detailed by its associated sequence diagram. In special the sequence diagrams excerpts represented by Figures 2.4b and 2.4c refine the "System identifies situation". Thus the reliability value assumed by the activity and represented by the variable *rSituation*, depends directly on the reliability computed for both sequence diagrams of Figures 2.4b and 2.4c. The way how such a variable is defined and computed will be shown later, by Sections 3.3 and 4.4.

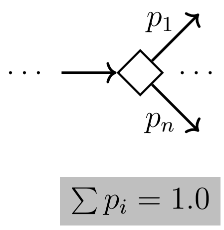


Figure 3.3: Decision node of a UML Activity Diagram

Decision node: the decision node shown by Figure 3.3 is used to represent alternative behaviors such an alternative is chosen based on the runtime verification of the software's state and context. The decision node has one incoming edge and as many outgoing edges as needed. Albeit the alternative choice is based on the runtime software's state the probability indicating how often each alternative is taken is defined by the domain expert *a priori* by assigning probabilities to each p_i variable in Figure 3.3. Finally, as each alternative has its execution probability the decision node has an associated constraint that the probability values of all outgoing edges must sum up to 1.0 — so it fulfills the basic property of DTMCs.

In the case of the running example presented by Section 2.5 the decision node "*Was there any QoS goal change?*" (c.f. Figure 2.4a) has the alternatives of executing the reconfiguration activity in case of a new QoSGoal otherwise, it simply bypasses such an activity. In such a case the domain expert has assigned 0.5 as the probability to each alternative.

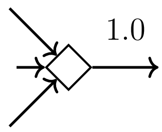


Figure 3.4: Merge node of a UML Activity Diagram

Merge node: the merge node is used to represent where several behavioral branches meet and the software execution proceeds into a single flow. As shown by Figure 3.4 a merge node has as many incoming edges as the number of branches being unified and an unique outgoing edge. Similar to its counterpart decision node, there is no software components interaction associated to this element. Therefore, as soon as the alternatives behaviors are merged, the software execution flows immediately to the next activity diagram element, as it is indicated by the 1.0 probability value of the outgoing edge. In the case of the running example the merge node represented in Figure 2.4a unifies the two behavioral branches created by the decision node into a single flow from it.

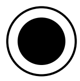


Figure 3.5: End node of a UML Activity Diagram

End node: the end node is used only once to define when the execution of the activity diagram is finished and it is represented by the surrounded filled circle shown by Figure 3.5. It has as many incoming edges as the number of behavioral branches having an activity diagram element considered final for that branch. Similar to its counterpart initial node, there is no software components interaction associated with the end node. In the case of the running example, the end node represented in Figure 2.4a denotes the control loop of the BSN-SPL has reached its end and can be executed again.

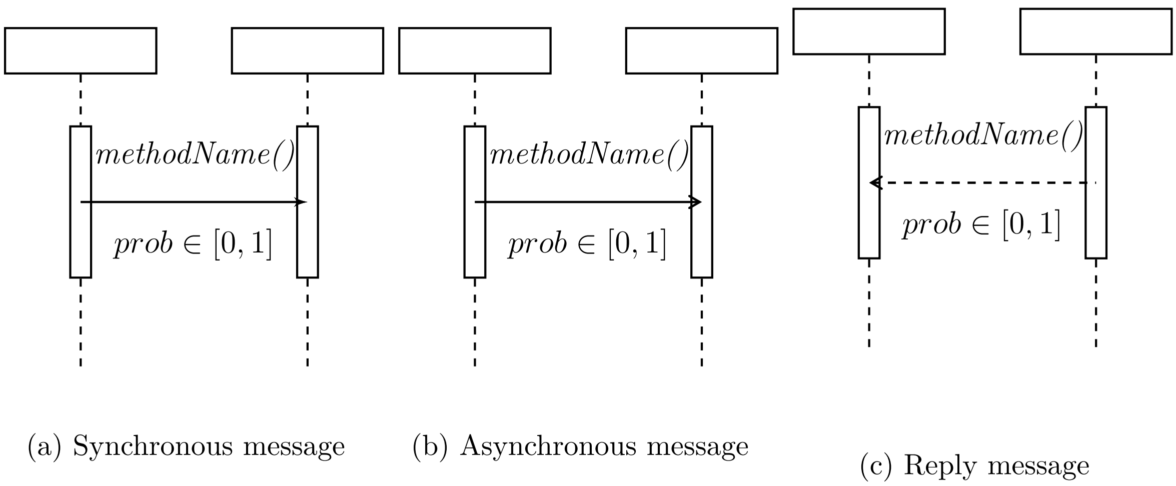
The set of activity diagrams elements considered in this work is sufficient for representing how the software product line behaves in the activity level. In such level it is not considered that a product have multiple and parallel or interleaved execution flows of its activities. Thus, two elements commonly used for representing multiple execution flows, namely *fork* and *merge* nodes, are not considered in this work.

3.1.2 UML Sequence Diagrams

The fine-grained behavioral modeling represents how the software's behavior is defined by the interactions among its software components. A *software interaction* comprises of two components, the method call, its execution mode and *reliability*. By reliability in this representation level, it is understood the probability of executing the method call between software components without errors occurrences. In addition, the fine-grained modeling must also represent iterative behaviors and alternative behavioral branches that may occur at runtime.

In the context of behavioral modeling of software product lines it is necessary representing both the behavior shared and the behavior specific to a set of products. The capability to differentiate in a diagram the common from the specific behaviors is what gives power to employ the UML sequence diagram to address the kernel of a software product line and to represent its whole behavioral variability. However, the semantics of a sequence diagram element must be adapted to accomodate the variability representation.

The jointly use of UML sequence diagram and UML MARTE profile allows modeling the variable and probabilistic behavior of a software product line by a detailed and fine-grained representation. The probability value assigned by UML MARTE's elements to a message between software components is sufficient to represent the communication channel's reliability. The sequence diagram elements used for representing the software product line's behavior are the *synchronous*, *asynchronous* and *reply* messages, besides the *alternative*, *loop* and *optional* combined fragments. Each element is described in the following.



(a) Synchronous message (b) Asynchronous message (c) Reply message

Figure 3.6: Messages types of a UML sequence diagram

From the structural point-of-view the synchronous, asynchronous and reply messages are equals. They are defined by an interaction between two software components (a.k.a. *lifelines*) that represents a method call between them. The message is represented by an arrow with its head varying as the message type, the method name placed over the arrow and its associated probability represented by the value assigned to the `prob` tag. Such value represents the communication channel's reliability.

Synchronous message: it is described in a sequence diagram by a solid and closed arrow between two lifelines as shown by Figure 3.6a. The synchronous message is used to represent a communication between two components in which the caller halts its execution and waits for the answer to be provided by the called component. As the caller component waits for the answer it is necessary having an associated `reply` message to each synchronous message used in the model. In the running example, the `register` and `persist` messages represented in Figure 2.4b are synchronous messages.

Asynchronous message: it is described by a solid and open arrow between two lifelines, as shown by Figure 3.6b. The asynchronous message is used to represent a communication in which the caller sends a signal to the called component but does not wait for the return. Thus, the asynchronous message does not have an associated reply message. In the running example, the `sendSituation` message represented in Figure 2.4b is an asynchronous message.

Reply message: it is represented by a dashed and open arrow directed to the caller lifeline as shown by Figure 3.6c. The reply message is used to represent the called method finished its execution and both control and result are returned to the caller method. The name placed over the arrow always starts with `reply` to reinforces the message is associated with a synchronous call. In the running example (c.f. Figure 2.4b) the messages `replyRegister`, `replyPersist` and `replySendSituation` are examples of reply messages.

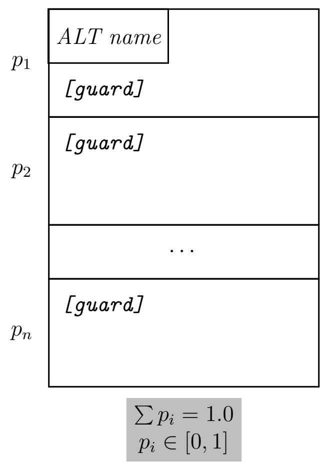


Figure 3.7: Alternative fragment of a UML sequence diagram

Alternative fragment: it is represented by a rectangle with the `ALT` tag placed at the top left corner, in addition to various inner rectangles called *lanes* as shown by Figure 3.7.

The alternative fragment is used to represent runtime behavioral variability whose decision is taken based on the runtime software's context. Each lane comprises several sequence diagram's elements defining its associated behavior, which is guarded by a propositional condition placed inside square brackets. Each lane's guard condition is verified at runtime and, in case it is satisfied, its comprised behavior is executed. Although the lanes' guard condition is only verified during runtime due to its dependency to the software's context, the domain especialist defines *a priori* the execution probability for each lane. Therefore, each lane has an associated probability $p_i, 1 \leq i \leq n$ tag whose value represents its execution probability such that the sum of all lane's probability must be equals to 1.0.

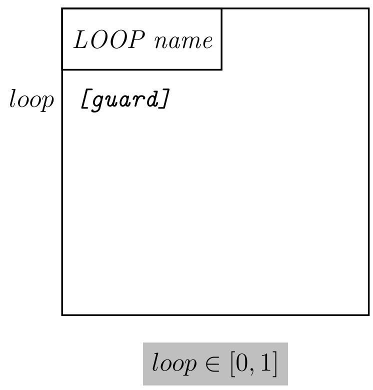


Figure 3.8: Loop fragment of a UML sequence diagram

Loop fragment: it is represented by the rectangle shown by Figure 3.8 with the `LOOP` tag placed at the top-left corner and the behavior depicted in its inside. The loop fragment is used to represent some behavior that repeats a given number of times. The number of iterations depends on the runtime evaluation of the fragment's guard condition that is represented by the propositional statement placed inside the square brackets. Similar to the alternative fragment, the domain expert must define its execution probability by assigning a value to the *loop* tag. Obviously, the probability of not executing the loop fragment assumes the complement of $1 - loop$.

Optional fragment: it is represented by a rectangle containing the tag `OPT` at its top-left corner followed by its name, as depicted in Figure 3.9. In its original semantics, it represents the behavioral variability that occurs in runtime, e.g., it can be used to model the behavior of an `if` conditional statement. However, such an element plays a major role in the context of modeling the behavioral variability of software product lines since such variability can happens in different times – not only during runtime). The optional fragment is the element responsible to represent, in an uniform manner, *all* the possible behavioral variability of a software product line because it relates the fragment's

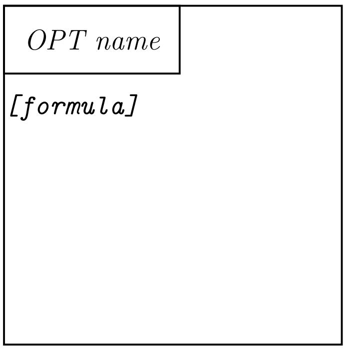


Figure 3.9: Optional fragment of a UML sequence diagram

behavior to the presence or absence of features in a set of configurations. The behavior is defined by the UML sequence diagram's elements comprised inside the fragment and the guard condition is expressed by a propositional formula inside square brackets whose atoms are named as the feature's names. Such a formula indeed represents the *presence condition* that must be fulfilled by configurations so that they execute its represented behavior. Thus, in a brief, such an element had its semantic changed in this work in order to (a) represent all kinds of variability by (b) enclosing the variable behavior in optional fragments (c) whose guard's formula will denote the set of partial configurations for which it will be considered. The configurations that do not comprise its satisfiability set will not present its behavior.

In the case of the running example the sequence diagrams shown by Figures 2.4b and 2.4c represent behavioral variability points related to the activity "System identifies situation". In Figure 2.4b the outermost optional fragment is associated to the optional *Oxygenation* feature (cf. Figure 2.3) since its guard condition is the atom `Oxygenation`. By its turn, such an optional fragment also has two variability points related to data persistence. The first fragment is associated to the *SQLite* feature by the atom `SQLite` and the second fragment is associated to the *Memory* feature by the atom `Memory`. Note that both *SQLite* and *Memory* are alternative features, but they are represented uniformly by optional combined fragments. As the behavior of the *Temperature* feature is similar to the behavior of *Oxygenation* feature, the rationale above also applies to the sequence diagram depicted by Figure 2.4c.

In summary, the UML activity and sequence diagrams can be used to represent the behavioral characteristics of a software product line. On the one hand, activity diagrams are used to represent the major tasks and execution flows that all instantied products must perform. On the other hand, sequence diagrams represent the shared and variable behavior, such a sequence diagram is a behavioral detailing of an activity. Both diagrams are enriched with the UML MARTE profile in order to allow them representing probabilistic behavioral information.

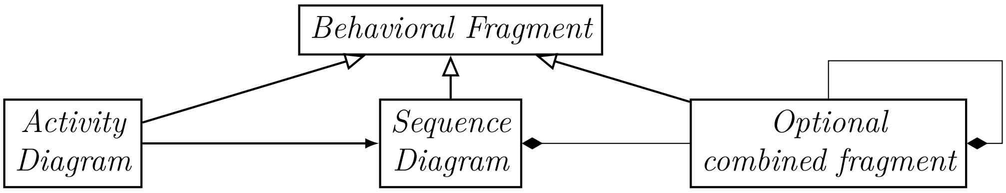


Figure 3.10: Associations between behavioral fragments, sequence diagrams and Optional combined fragments

Finally, it worth describing the relationship among the UML behavioral diagrams considered in this work. Such relations are represented by Figure 3.10. Any kind of behavioral representation is considered a *behavioral fragment* which can be specialized into an activity diagram, sequence diagram or an optional combined fragment. The association from activity diagram and sequence diagram represents the refinement relation between an activity and its associated sequence diagram. In addition, a sequence diagram can be comprised of several optional combined fragments for representing its behavioral variability. By its turn, an optional combined fragment has its behavior represented by a sequence diagram which, by its turn, can have variability points described by other optional combined fragments. Such relations are important for the evaluation method due the steps of the evaluation method considers and explores the polymorphism between such elements when evaluating the reliability of each behavioral fragment.

3.2 Reliability of UML Behavioral Models

Intuitively, the software reliability computed from a UML behavioral diagram is the execution probability of all its possible behaviors from the first until last element, such that the software behavior is defined by a set of execution sequences of actions or methods (a.k.a. *paths*). Thus the reliability analysis implies into identifying and computing the probabilities of all possible executions represented in the diagram. Since each possible execution in a behavioral diagram is a sequence of UML behavioral elements, the probability associated to each element placed along the path must be considered when computing its probability.

A path in a behavioral diagram consists of a finite sequence of behavioral elements such that one element is only executed just after its previous element finishes its execution. The order which such elements are disposed is temporal and defined according to the execution manner of the behavioral diagram. Formally, an execution path is a partial order defined as:

$$\varrho = e_1\alpha_1e_2\alpha_2\ldots\alpha_m e_n \tag{3.1}$$

where ϱ is the execution path; $e_i, 1 \leq i \leq n$ is each behavioral element comprising the path ϱ and $\alpha_j, 1 \leq j \leq m$ is an action linking two behavioral elements. Since a path is a *serial* execution of behavioral elements, its reliability is computed as the product of all elements's reliabilities. An element in a given place along the path has its *accumulated reliability* that is computed by multiplying the element's probability and the accumulated reliability of its next element. Formally, the accumulated reliability of an element e is given by the recursive function

$$R(e_i) = prob(e_i) \times R(e_{i+1}) \tag{3.2}$$

where $prob(e_i)$ is the probability associated to e_i and $R(e_{i+1})$ is the accumulated reliability computed for its direct successor. Thus, considering the accumulated reliability is computed in a recursive fashion, the path's reliability is given by the accumulated reliability of its first element, which is defined as:

$$R(\varrho) = R(e_1) \tag{3.3}$$

where $R(\varrho)$ is the reliability of the path ϱ and $R(e_1)$ is the accumulated reliability of the first path's element. However, the reliability definition varies according to the semantics of each behavioral element so that the reliabilities definitions of all behavioral elements considered in this work are shown in Sections 3.2.2 and 3.2.3.

Finally, given the relations of behavioral fragments represented by Figure 3.10 a general definition for the reliability computation of a behavioral fragment must be useful to compute the reliability of activity diagram, sequence diagram and optional combined fragment in an uniform way. According to Sections 3.1.1 and 3.1.2, both activity and sequence diagrams have an unique starting point (the start node and the first behavioral element, respectively). Thus, given that paths in activity and sequence diagrams are sequences of behavioral elements (cf. Definition 3.1) and an optional combined fragment comprises a sequence diagram in its inside, indeed the reliability definition for a behavioral fragment can be generalized and formally defined as

$$\begin{aligned} R(bf) &= R(\varrho) \\ &= R(e_1) \end{aligned} \tag{3.4}$$

where ϱ is the path whose first element is the unique starting element of the behavioral fragment bf.

Therefore the reliability of a behavioral model is computed in an inductive fashion based on the structure of the UML behavioral model. Considering the software product line's behavior is expressed by a set of UML behavioral models, its reliability is given by each behavioral model's reliability and the way such models are related. On the one hand the reliability of the whole software product line is defined by the manner how the activities are related in an activity diagram and how each activity is detailed by a sequence diagram. At the other hand, since an activity is detailed by its associated sequence diagram its reliability is also associated to the reliability computed for its sequence diagram. Such reliabilities notions are explained in details next.

3.2.1 Reliability of software product line

To compute the reliability of a software product line it is necessary to consider the reliabilities of all behavioral diagrams and the manner how such diagrams are related. By associating a sequence diagram to each activity it is possible to define the set of all behavioral diagrams as well as how such sequence diagrams are related. Thereby, the reliability of a whole software product line is the reliability of its UML activity diagram, which can be formally defined as

$$R(SPL) = R(SPL.ad) \tag{3.5}$$

where SPL is the software product line subject of the evaluation and $SPL.ad$ is the activity diagram representing the coarse-grained behavior of the SPL.

3.2.2 Reliability of activity diagram elements

According to Definition 3.5, the software product line's reliability is computed based on its activity diagram, which by Definition 3.4, is the accumulated reliability of its first element (i.e. the product of all elements' reliabilities along the path). Since the reliability function R is inductive in the structure of the behavioral model it is necessary to compute the accumulated reliability for each element along the path which is given by the element's probability value multiplied by the accumulated reliability of its immediately successor. Next, the accumulated reliability definition for each activity diagram's element is presented. From now on it is meant by behavior any interaction between software components.

Initial node: since the initial node does not have an associated behavior its impact over the reliability computation is null. Thus, its accumulated reliability is equal to 1.0

multiplied by the accumulated reliability of its remaining path. Formally it is defined as

$$R(in) = 1.0 \times R\Big(next(in)\Big) \tag{3.6}$$

where in is the initial node and $next(in)$ is the singleton set of its immediately successor and $R(next(in))$ is the accumulated probability of such element.

Activity: due to an activity is detailed by a sequence diagram its reliability varies according to such diagram's behavior and the product instantiated. Thus, it is agreed to represent the activity's reliability as a variable named as the activity with the 'r' prefix standing for "reliability". Formally it is defined as

$$R(a) = rActivity \times R\Big(next(a)\Big) \tag{3.7}$$

where a is an activity named $Activity$; $rActivity, rActivity \in [0, 1]$ is the variable representing the activity's reliability, $next(a)$ is the successor of a and $R(next(a))$ is the accumulated reliability of its immediately successor.

Decision node: the decision node is the element that splits the behavior into different paths such each alternative path has an associated execution probability defined by the domain expert (c.f. Figure 3.3). Given that each path is independently executed from the others, the reliability of the decision node is the sum of the cummulative reliabilities of all alternative paths. Such a reliability definition is formally given by

$$R(d) = \sum_{i=1}^{n} pr_i \times R\Big(next(d, i)\Big) \tag{3.8}$$

where n is the number of outgoing edges of the decision node d, pr_i is the probability associated to the i^{th} edge, $next(d, i)$ is the singleton set of the immediately successor of d by the edge i and $R(next(d, i))$ is the accumulated probability of such element.

Merge node: the merge node joins different behavioral branches into a single path without any interaction between software components. Thus, its impact over the reliability computation is null and its accumulated reliability is formally defined as

$$R(m) = 1.0 \times R\Big(next(m)\Big) \tag{3.9}$$

where $next(m)$ is the immediately successor of m and $R(next(m))$ is the accumulated reliability computed for such an element.

End node: the end node determines the execution's end of an activity diagram without any associated behavior such that its contribution to the accumulated reliability is null. It is also the base case of the recursive function R when computing the reliability of an activity diagram. It is formally defined as

$$R(e) = 1.0 \tag{3.10}$$

where e is the end node.

3.2.3 Reliability of sequence diagram models

The reliability computation of a sequence diagram resembles the reliability computation of models described by activity diagrams. For both the reliability is given by the accumulated reliability of the first element (as stated by Definition 3.4) and the path's reliability is inductively computed on the path structure by the product of all elements in its sequence (as stated by Definition 3.2). However, the sequence diagram's representation structure and the need of representing the behavioral variability turn the reliability function R for sequence diagram slightly different from the reliability function of activity diagram.

The first difference stems from the representation structure of a sequence diagram: differently of an activity diagram whose elements are linked by transitions, there is no link between the elements comprising a sequence diagram. The unique relation between the sequence diagram's elements is the point where each element is represented in the diagram which, given its top-down reading way, imposes a temporal sequence for its elements. Therefore, to compute the accumulated reliability of a sequence diagram element it is necessary to consider which elements may execute in the next step.

In addition, the sequence diagram is the responsible for representing the software product line's behavioral variability as some behavioral fragments will comprise a set of products but will not in another set. Given the accumulated reliability of an element is a recursive function, such function must be able to consider such variability of the software product line in an uniform manner.

Thus the reliability function R for sequence diagrams has to deal with the temporal link between the elements of a variable behavioral diagram. Both characteristics impose a semantic change of a sequence diagram element for representing the inherent variability of a software product line and also consider the sequence of the sequence diagram elements as a partial-ordering. The definitions of reliability function R for the sequence diagram's elements are presented next.

Synchronous, asynchronous and reply messages: albeit the execution of synchronous, asynchronous and reply messages have different execution modes (c.f. Section 3.1.2), they are similar from the structural and reliability points of view: all of them are a communication between two software components that is accordingly performed bounded by the communication channel's reliability. Thus the accumulated reliability for such messages is given by the probability associated to the message multiplied by the accumulated reliability of its next element. Such a definition is formally given by:

$$R(m) = prob(m) \times R\Big(next(m)\Big) \tag{3.11}$$

where $prob(m)$ is the function that returns the probability value associated to the message m and $next(m)$ is the function that returns its next executable element and $R(next(m))$ is the cummulative reliability computed for the successor of m.

Loop fragment: Since the loop fragment has a probability value defined by the `loop` variable for the cases it is executed (consequently its complement represents the probability of not executing the loop), such a value must be considered for its reliability definition. In the case the loop's behavior is executed, its inner content's reliability must also be taken into account, otherwise the non-execution probability must be considered. Such inner behavior is, indeed, represented by a sequence diagram associated to the loop fragment. Thus, the reliability of a loop fragment is given by the accumulated probability of both loop's execution and non-execution. Such a reliability is formally defined as

$$R(l) = \Big(prob(loop) \times R\Big(SD(l)\Big) + \Big(1 - prob(loop)\Big)\Big) \times R\Big(next(l)\Big) \tag{3.12}$$

where $prob(loop)$ is the fragment's execution probability, $SD(l)$ is the function that returns the sequence diagram of the loop l, $R(SD(l))$ is the reliability computed for such diagram (i.e. the loop's innerbehavior), $1{-}prob(loop)$ is the probability of not executing l, $next(l)$ is the directly successor of l and $R(next(l))$ is the accumulated reliability of the immediately successor of l.

Alternative fragment: as the alternative fragment represents the behavior splitting into several alternatives, in practice it creates several possible execution flows from it such that each alternative has its execution probability defined by the domain expert. Besides, since each alternative's behavior is represented by a sequence diagram, its reliability is given by the reliability computed for its associated sequence diagram. Thus, the reliability of each alternative is given by its choice probability multiplied by the reliability computed for its sequence diagram.

Whatever branch is taken, the path continues from the element just after the alternative fragment. The cummulative reliability of the alternative fragment must consider the reliability of the path remaining after it, that is computed as the cummulative reliability of its direct successor. Thus, the cummulative reliability of the alternative fragment is formally defined as:

$$R(a) = \Big(\sum pr_i \times R\big(SD(alt_i)\big)\Big) \times R\big(next(a)\big) \tag{3.13}$$

where pr_i is the execution probability associated to the i^{th} alternative, $SD(alt_i)$ is the sequence diagram of the i^{th} alternative, $R(SD(alt_i))$ is the reliability computed for the sequence diagram of the i^{th} alternative and $R(next(a))$ is the cummulative reliability computed for the successor element of the alternative fragment.

Optional fragment: the optional behavioral fragment represents the behavioral variability of software product lines according to the presence or absence of features in a configuration. An optional fragment will always be considered by the paths passing through it. But it is necessary to define when its reliability must be considered or not, such that it will not impact the computation. The accumulated reliability for the optional fragment is thus computed based on the fragments' behavior presence or absence, in addition to the accumulated reliability computed from its next element. Formally it is defined as:

$$R(o) = R\big(SD(o)\big)^p \times R\big(next(o)\big) \tag{3.14}$$

where $SD(o)$ is the sequence diagram associated to the optional combined fragment o, $R(SD(o))$ is the reliability computed for the sequence diagram associated to o, $p \in \{0, 1\}$ indicates the fragment presence (1) or absence (0) and $R(next(o))$ is the accumulated reliability of the fragment's successor.

In the case of the running example, both sequence diagrams shown by Figures 2.4b and 2.4c have their behaviors varying according to $SQLite$ and $Memory$ features because they are related to the fragments `SQLite` and `Memory`, respectively. In addition both features are alternative according to the feature model represented in Figure 2.3. Thus, whenever one of the persistence features is present in a configuration the other one is necessarily absent such that its behavior must not affect the reliability computation. In the case of $SQLite$ is part of the configuration the p exponent in its term assumes the value 1, meanwhile the p exponent of the term related to $Memory$ feature assumes the value 0 to represent its absence. Thus, the reliability of the `SQLite` fragment is given by

$$\begin{aligned} R(SQLite) = & R\Big(SD(\texttt{SQLite})\Big)^{1} \times R\Big(next(\texttt{SQLite})\Big) \\ & R\Big(SD(\texttt{SQLite})\Big) \times R\Big(SD(\texttt{Memory})\Big)^{0} \times R\Big(\ next(\texttt{Memory})\Big) \\ & R\Big(SD(\texttt{SQLite})\Big) \times 1 \times R\Big(\ next(\texttt{Memory})\Big) \end{aligned}$$

which, indeed, only considers the reliability of the fragment associated to $SQLite$ feature. Conversely, the same rationale holds for the cases where *Memory* is present and *SQLite* is absent in the configuration.

3.3 Transformation from UML to FDTMC

Once the whole behavior of a software product line is represented by a set of UML behavioral diagrams it can be transformed into fully probabilistic models that later will be subject to reliability analysis. However, given the expressiveness of UML behavioral diagrams (messages execution modes in addition to alternative, iterable and optional behaviors) such characteristics must also be considered by the probabilistic models. In this context, the FDTMC is a suitable modeling notation as it is able to represent the variable and probabilistic behavior of a software product line.

The UML behavioral models present an *action-based* view of the software product line's behavior because they represent how the variable behavior is defined by the behavioral elements and executed by the software components. Meanwhile, FDTMC provides the corresponding *state-based* view since it represents the behavioral characteristics by expressing how the software product line's states vary as long as its behavior is executed. In an FDTMC, each state represents the observable characteristics reached by the software after executing its respective UML element.

Given the dependency between the action- and state-based views, a set of translation rules T to an FDTMC sub-structure is defined for each UML behavioral element. A transformation rule is defined by two parts: the left-hand side shows the UML element being transformed and the right-hand side shows its resulting FDTMC sub-structure. From now on assume that elements depicted by solid lines at the right-hand side are the new FDTMC's states or edges created by the transformation rule, while the elements represented in dashed lines already exist. Assume also that the name *current state* is the state from which the FDTMC substructure will be built. Thereby, a whole UML behavioral model can be translated into its respective FDTMC by applying the transformation rule of each behavioral element in a stepwise fashion. Next the translation rule and the resulting FDTMC for each UML behavioral element is presented jointly with the description of its state-based view.

3.3.1 Transformation Rules for Activity Diagram Elements

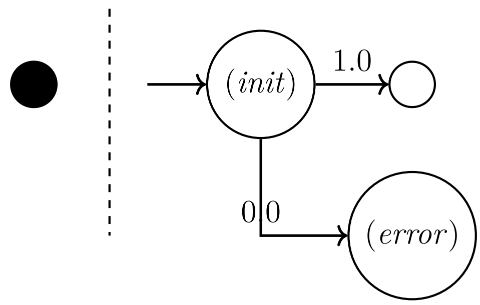


Figure 3.11: Transformation rule for an initial node of an activity diagram

Initial node: the resulting FDTMC shown by Figure 3.11 is comprised of three states and two edges. The labeled states `init` and `error` represent, respectively, the state where the execution begins and the state representing some error occurred during the diagram execution. The edge from `init` to the unlabeled state has the associated probability value 1.0 and, due to the basic FDTMC's property, its complement edge has 0.0 as probability value.

The state-based view provided by the resulting FDTMC means the execution starts (`init` state) and proceeds with probability equals to 1.0 to the next state. Such a value follows from the fact that there is no software behavior associated to the initial node, so its failure probability is equal to 0.0.

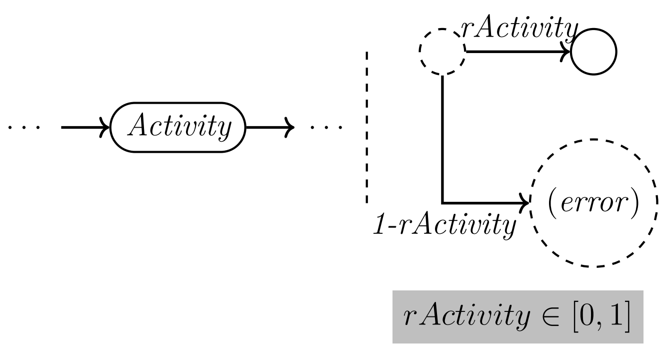


Figure 3.12: Transformation rule for an activity node of an activity diagram

Activity: intuitively, the reliability of an activity is the reliability computed by its associated sequence diagram. The activity's transformation rule represented by Figure 3.12 comprises a new state and two new edges. The edge directed from the current to the newly created state has its probability represented by the *rActivity* variable, $rActivity \in [0, 1]$, whereas the edge directed to the error state assumes its complement $(1 - rActivity)$.

The current state represents the moment just before the activity execution. With a probability equals to the value assumed by *rActivity*, its associated behavior is performed without errors and such a condition is represented by the newly created state. Following the naming convention adopted by the UML behavioral modeling (c.f. Section 3.1.1) the edge's parameter is named as the activity's name with the prefix 'r' standing out for "reliability". In case of an error occurred anywhere of the sequence diagram associated to the activity, such a condition is represented by the error state that is reached with the probability given by $1 - rActivity$.

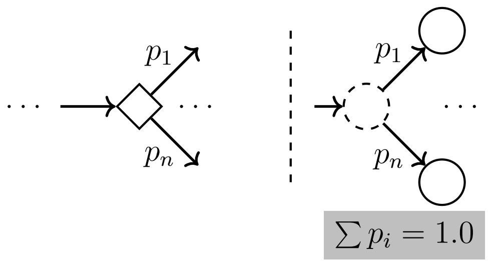


Figure 3.13: Transformation rule for a decision node of an activity diagram

Decision node: the translation rule shown by Figure 3.13 results into an FDTMC comprised of as many newly edges and states leaving the FDTMC's current state as the alternatives of the decision node.

The state-based view provided by the FDTMC means the software execution may proceed from the current state to the first state of each alternative with the probability associated to its respective edge. The current state represents the software state just before the decision is taken and each newly created state represents the software state just before its behavioral branch starts its execution. As each decision of a decision node has a probability value assigned by the domain expert, such probabilities are considered when creating each edge leaving the current state of the FDTMC. Given the basic property of FDTMCs the probabilities of all edges leaving the current state must sum 1.0. Finally, as there is no software behavior associated to the UML decision node, there is no edge from current to the `error` state.

Merge node: the resulting FDTMC shown by Figure 3.14 is comprised of several newly created elements namely: two states, an edge between both states with probability

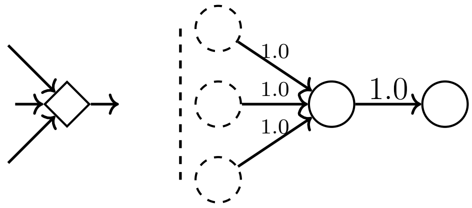


Figure 3.14: Transformation rule for a merge node of an activity diagram

equals to 1.0 and as many edges as the number of existing states to be merged. Such edges also have 1.0 as transition's probability.

The FDTMC's state-based view represents by each current state the execution's end of an alternative behavioral branch while the first newly created state represents the merging state, ie., the software's state where all alternatives are ready to merge. From the merging state, the execution proceeds to the merged state (the second state created by the transformation rule) with probability equal to 1.0 because there is no software components interaction enrolled in this task.

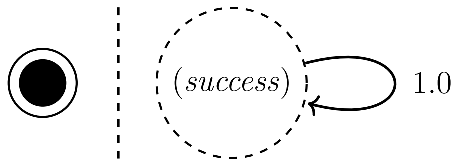


Figure 3.15: Transformation rule for an end node of an activity diagram

End node: the transformation rule shown by Figure 3.15 results into an FDTMC comprised of a newly created self-edge with probability equals to 1.0 and a new label placed at the current state.

The state-based view of the resulting FDTMC represents the activity diagram execution has ended successfully due the execution has reached the final "*success*" labeled state. As the success state has no associated components interaction the self-edge assumes the 1.0 probability value that transforms it into an absorbing state.

3.3.2 Transformation Rules for Sequence Diagram Elements

The UML sequence diagram's elements represent the small constituents parts of a software and the way they are arranged defines the software's behavior. The transformations from sequence diagram elements into FDTMC structures resemble the activity diagram's transformations and a transformation rule is defined for each sequence diagram element. Again, each transformation rule represents the newly created edge and states by solid lines meanwhile the already existing states and edges are represented by dashed lines. In the sequence, the transformation of each sequence diagram element is described with its state-based view.

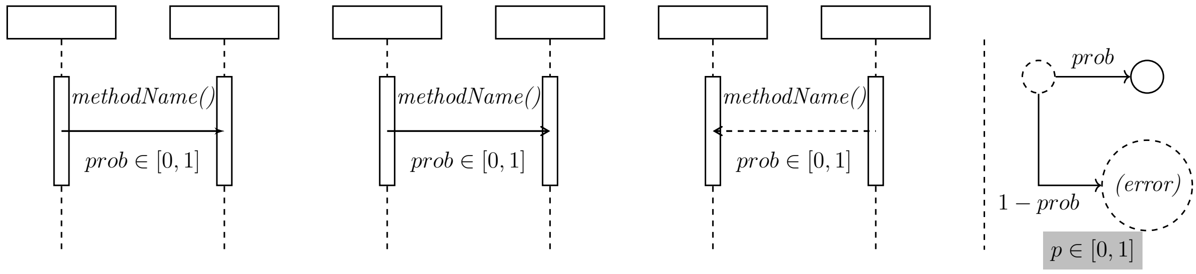


Figure 3.16: Transformation rule for a synchrounous, asynchronous and reply messages of a sequence diagram

Synchronous, asynchronous and reply messages: albeit each kind of message has its meaning well defined in the context of software modeling by UML sequence diagrams, the resulting FDTMC from their transformations are the same in terms of number of nodes and edges. The differences are centered in the meaning of some states. Thus, the resulting structure for such messages and such differences are explained below.

The transformation rule for all messages (synchronous, asynchronous and reply) results into an FDTMC comprised of two already existing states, and newly created edges and state, as shown by Figure 3.16. The edge from the current to the newly created state has the probability $p, p \in [0, 1]$ while the complement edge directed to the `error`-labeled state has the $1 - p$ probability.

For all kinds of messages, the FDTMC's state-based view represents by the current state that the caller component is ready to place the method call. Also for all kinds of messages the error state represents an error occurred during the method call whose probability is given by the value of the complement edge $(1 - p)$. The edge from the current to the newly created state represents for all messages types the probability p of sending the message without any error occurrence. However, the meaning of the newly created state varies according to the message type. In the case of a synchronous message it represents the caller component is halted meanwhile the called component is executing. In the case of an asynchronous message, it represents that both caller and called components are executing their behavior just after placing the method call. Finally, for the case of reply message, it represents the called method ended its execution and the control is back to the caller component.

Alternative fragment the transformation rule shown by Figure 3.17 considers the probabilities assigned by the domain expert to each lane comprising the activity diagram.

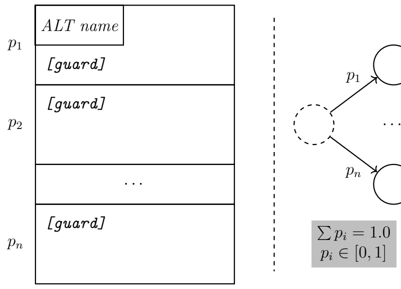


Figure 3.17: Transformation rule for an alternative fragment of an sequence diagram

The resulting FDTMC is comprised of the current state with as many newly created outgoing edges and states as the number of lanes. Each edge assumes its probability value according to its respective lane represented in the alternative fragment, such that $p_i \in [0, 1], \sum_{i=1}^{n} p_i = 1.0$.

The FDTMC's state-based view represents the moments just before and after the choice of the behavioral branch. The current state represents the software's context at the choice moment while each of its imediately successor states represent the software context just before starting the execution of the behavior of the choosen branch. The edge's probabilities of the FDTMC assume its respective probabilities given by the domain expert at the alternative fragment. Due the alternative choice does not involve any interaction between software components, ie. a single component makes the decision about which branch must be taken according to the software's context, there is no failure probability so there is no edge from the current to the `error` state at the FDTMC.

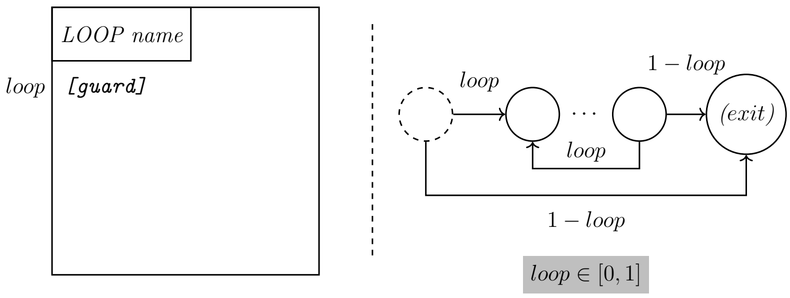


Figure 3.18: Transformation rule for a loop fragment of a sequence diagram

Loop fragment: the FDTMC resulting from the transformation rule shown by Figure 3.18 is comprised of 3 and 4 newly created states and edges respectively, in addition to the current state. The current state transits to the second state with the probability given by the parameter `loop` defined by the domain expert, or it transits to the `exit` labeled state with the complementary probability `1-loop`. The analogous reasoning applies to the third state. The ellipsis between the second and third states abstracts the resulting FDTMC from the transformation of loop's content.

The current state represents the software's context whose runtime evaluation will decide whether the loop fragment will be executed. The first inner state represents the initial state of the loop's content. The second inner state indicates the loop's content had executed and another decision about the iteration must be taken. In case it has to be executed again, the execution proceeds to the first inner state. Otherwise, it leaves the loop and proceeds to the first state after the loop content, ie. the `exit` labeled state. Such state represents the software is ready to execute the first action just after the loop fragment.

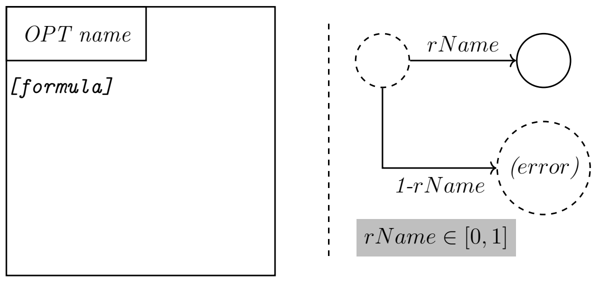


Figure 3.19: Transformation rule for an optional combined fragment of a sequence diagram

Optional fragment the transformation rule results into the FDTMC shown by Figure 3.19 that is comprised of three states and two edges, such a state and the edges are newly created by the rule. The current state transits to the new state with the probability $rName, rName \in [0,1]$ while its complement edge assumes the probability $1 - rName$.

The current and the newly created states represent, respectively, the software's context just before and after the optional fragment execution. Due to the resulting FDTMC represents the optional fragment's reliability, it is agreed the edge's variable is named as the fragment's name with the 'r' prefix standing for reliability. The `rName` variable represents the reliability computed for the entire optional combined fragment in case its behavior is present (ie., for the cases its guard condition is satisfied). Hence the complement edge represents the failure probability for executing the optional fragment. In particular, when the guard condition is not fulfilled by a configuration, the optional fragment's behavior is not considered part of the product and the `rName` parameter assumes the value 1.0. By assuming such value the complement edge assumes the probability value 0.0 that represents there is no failure probability. Indeed, when the `rName` assumes the probability value 1.0 it means there is no software behavior associated to the edge, thus the optional combined fragment has no effect at the reliability analysis.

3.4 Reliability Equivalence of UML Behavioral Models and FDTMCs

Given a set of UML activity and sequence diagrams representing the software product line's behavior, it is possible to compute its reliability by two distinct manners: a) it can be computed by transforming the UML models into their respective FDTMCs and then employing state-of-the-art probabilistic model checkers or b) by applying the reliability functions of each behavioral element (cf. Section 3.2) in a stepwise fashion and then traversing the resulting derivation tree solving for a given configuration. However, the later evaluation alternative lacks of tools implementing it whereas, in theory, any parametric and probabilistic model checker is able to evaluate FDTMCs. Thus, in order to allow the reuse of existing and state-of-the-art parametric model checkers, it is necessary to demonstrate some evidence for the equivalence between reliabilities computed from both UML and FDTMC models.

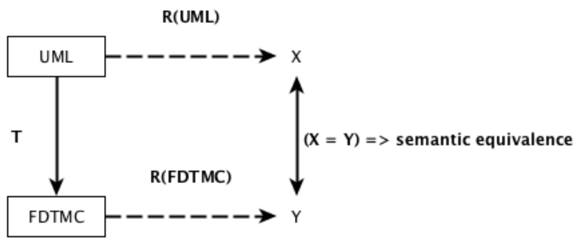


Figure 3.20: Intuition of the reliability equivalence of UML and FDTMCs models

Providing a formal proof of the equivalence for reliabilities computed for both UML and FDTMCs is out of the scope of this work. However some evidences and arguments presented in the following show the reliability formulae computed from both models are equivalent. The intuition for such a demonstration is depicted by Figure 3.20. The elements in boxes represent behavioral models of a software product line. Given a UML behavioral model it is possible to create its respective FDTMC by applying the set of transformation rules T presented in Section 3.3 that are represented by the leftmost arrow. Both dashed arrows leaving the behavioral models represent a possible reliability evaluation for the software product line. The top arrow represents the reliability evaluation of UML behavioral models by the stepwise application of the reliabilities definitions of Section 3.2. Such an evaluation results into a derivation tree whose terminal nodes denote constants and terms of the reliability formula such that the variables represent the reliabilities of optional combined fragments. The bottom edge represents the reliability evaluation of the FDTMCs resulting from the translation rules applied at the UML behavioral models. Such an evaluation employs algorithms defined for parametric model checkers[41] resulting into a parametric formula whose parameters represent the reliabilities values its respective FDTMC may assume. Thus, to evidence the equivalence of both reliability evaluations it is necessary demonstrate that the computed reliabilities must be the same when evaluating both formulae from their first to their last element (state in the case of FDTMCs and element in the case of the UML behavioral models).

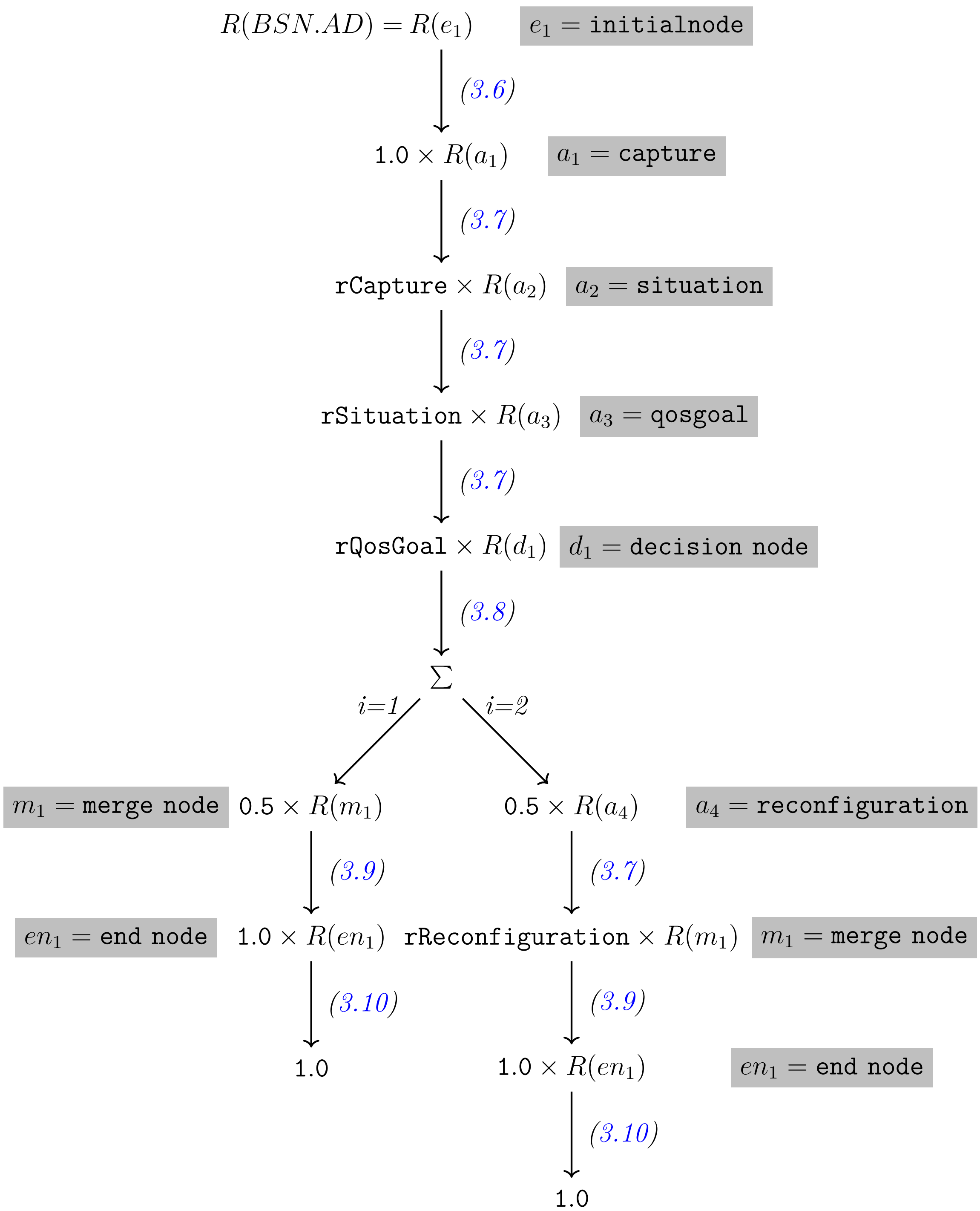
In the next subsections the reliability equivalence of both UML and FDTMCs models will be demonstrated, using the behavioral models of BSN-SPL represented by Figures 2.4a and 2.4b.

3.4.1 Reliability equivalence for activity diagram

Intuitively, the reliability of a UML behavioral model is given by the probability of executing all its behavior from the first until the last element without any error occurrence. According to Definition 3.4 it is given by the accumulated reliability of its first element which, as stated by Definition 3.2, is a recursive function that considers the accumulated reliability of all elements along the path. Since each element has its own reliability definition (cf. Section 3.2.2), the reliability computation for a UML activity diagram consists into the stepwise application of the elements' reliability definitions. Such a computation results into a derivation tree that, when traversed in a pre-order fashion considering only terminal nodes and operators, results into its reliability formula.

Thus, for the case of the activity diagram of the BSN-SPL (represented by Figure 2.4a), its reliability can be computed by applying the reliability definitions presented in Section 3.2.2 until there is no more node to be expanded by a reliability definition rule, so the derivation tree is complete. For the sake of space, the stepwise application rule is not shown, but the whole derivation tree is represented in Figure 3.21. Each derivation step is represented by an edge linking two nodes containing the reliability definitions for activity diagram elements with the number of the applied definition rule placed aside. Finally, the element whose reliability will be computed in the next step (i.e. the element returned by the `next` auxiliary function) is shown as a comment in gray boxes.

At this point two remarks worth to be addressed regarding the derivation tree presented by Figure 3.21. Initially, it is known the activity diagram of the BSN-SPL (c.f. Figure 2.4a) has a common flow of activities until its decision node. Such a node splits the behavior into two execution flows for the cases a reconfiguration is or is not necessary. Such a split is indeed considered in the derivation tree by the reliability Definition 3.8 for decision nodes, such that from that point the derivation tree also splits in two branches. The second remark regards to the formula resulting from the tree traversal in a pre-order fashion. By considering the terms and operators of each node the reliability for such activity diagram is given by the formula $R(BSN.AD) =$



$$
\begin{aligned}
R(BSN.AD) = {} & 1.0 \times rCapture \times rSituation \times rQosgoal \times \\
& (0.5 \times 1.0 \times 1.0 + 0.5 \times rReconfiguration \times 1.0 \times 1.0)
\end{aligned}
$$

$$
R(BSN.AD) = rCapture \times rSituation \times rQosgoal \times (rReconfiguration + 1)
$$

Figure 3.21: Derivation tree and reliability formula computed for the activity diagram of the BSN-SPL

$1.0 \times rCapture \times rSituation \times rQosgoal \times$
$(0.5 \times 1.0 \times 1.0 + 0.5 \times rReconfiguration \times 1.0 \times 1.0)$ that, in its simplified form, is equal to $R(BSN.AD) = 0.5 \times rCapture \times rSituation \times rQosgoal \times (rReconfiguration + 1)$. In such a formula, the common flow of activities reflects into the multiplication of the four initial terms, meanwhile the decision node is the responsible for generating the sum of $(rReconfiguration + 1)$. As the decision node is part of the execution flow, its sum is multiplied by the multiplication related to the common flow (i.e. the three initial activities and the decision's node probability).

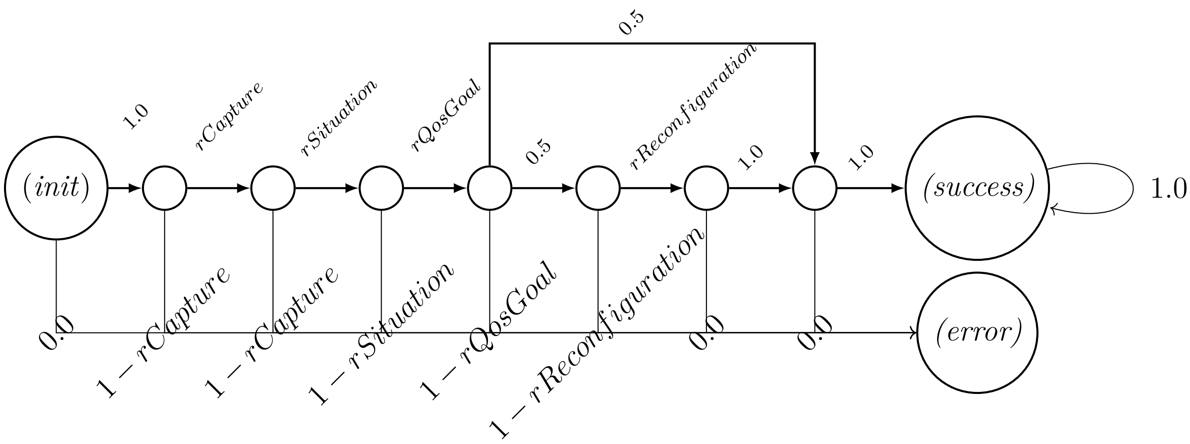


Figure 3.22: FDTMC of the activity diagram of the BSN-SPL

The other manner to compute the reliability of the activity diagram is to apply the transformations from UML to FDTMC substructures defined for activity diagram elements in Section 3.3.1 in stepwise fashion and then evaluate the resulting FDTMC by means of a parametric model checker. Since the reliability is the property of interest, it can defined as the probability of reaching the FDTMC's final state labeled as "success"by the PCTL statement $P_{=?}(\Diamond''success'')$. Again, for the sake of space the stepwise construction of the FDTMC is not shown, but the resulting FDTMC is shown by Figure 3.22. In such a FDTMC, it is possible to infer that only two possible paths lead to the success state. Both paths have the initial four transitions and, from the fifth state, the execution splits into two possible paths. Such splitting stems from the transformation rule 3.13 defined for decision nodes of activity diagrams. Later, such paths merge into a common flow at the 8^{th} state, as defined by the transformation rule 3.14. Albeit the transition from the `init` to the *error* state never occurs (since its probability is 0.0) it is represented at the Figure 3.22 to demonstrate the basic property of FDTMCs is fulfilled. According to the reliability definition provided by [23], the reliability can be computed by a rechability measure in a probabilistic model that, intuitively, is defined as the sum of probabilities computed for each possible execution path of a probabilistic model [10].

When a parametric model checker's algorithm[41] is employed to verify the aforementioned PCTL statement at the FDTMC shown by Figure 3.22 the resulting formula for its reliability is equal to $1.0 \times rCapture \times rSituation \times rQosgoal \times 0.5 \times 1.0 + 1.0 \times rCapture \times rSituation \times rQosgoal \times 0.5 \times rReconfiguration \times 1.0 \times 1.0$, where the variables starting with 'r' denotes the reliability of its related activity. Such a formula, in its simplified form, is equal to $0.5 \times rCapture \times rSituation \times rQosgoal \times (rReconfiguration + 1)$.

Albeit the UML activity diagram and its related FDTMC describes different characteristics of the BSN-SPL (the activity diagram provides the action-based view meanwhile the FDTMC the state-based view), the reliability formulae computed from both models are equals. Despite it is not a formal demonstration of reliability equivalence between such models, the equality of such formulae provides evidences that the reliability of a UML behavioral model can be computed by employing algorithms of parametric model checkers into its related FDTMC. In addition, such a demonstration also provides evidences that the transformation rules for activity diagram elements (cf. Section 3.3.1 are correct in the sense they preserve the reliability notion of the considered elements.

3.4.2 Reliability equivalence for sequence diagram

Once the demonstration presented in Section 3.4.1 provides evidences that there is equivalence of reliabilities computed from UML activity diagram and its related FDTMC, it is necessary obtain similar evidences for UML sequence diagram and its FDTMC. Thus, the demonstration's intuition represented in Figure 3.20 still holds and the notation used to represent the derivation tree will also be used for the reliability definitions of sequence diagrams.

$$R(SQLite) = R(m1) \qquad \boxed{m1 = \texttt{persist}}$$

$$\Big\downarrow (3.11)$$

$$0.999 \times R(m2) \qquad \boxed{m2 = \texttt{replyPersist}}$$

$$\Big\downarrow (3.11)$$

$$0.999$$

$$R(SQLite) = 0.999 \times 0.999$$

$$R(SQLite) = 0.999^2$$

Figure 3.23: Derivation tree of reliability definitions for the *Sqlite* feature

Figure 3.23 presents the derivation tree of the fragment `rSqlite` presented in the Figure 2.4b. Since the behavior of such a fragment is represented by its inner sequence diagram and such a diagram does not have any variability point, the demonstration related in the following also holds for all sequence diagrams that does not have variability points. Such a fragment comprises a synchronous and its reply message for the data persistence. Thus, the reliability Definition 3.11 is applied twice resulting into the derivation tree (Figure 3.23. Since the sequence diagram does not have variability points its resulting reliability formula is given in terms of constants. Thus, the reliability computed for such a fragment is $R(SQLite) = 0.999^2$.

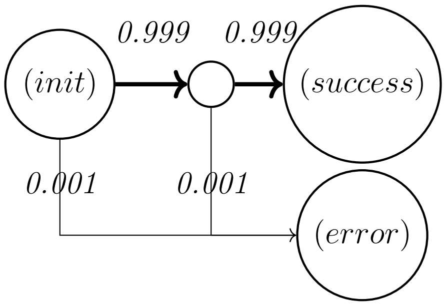


Figure 3.24: FDTMC of the *Sqlite* feature

In the case of the reliability analysis by means of a model checker, the FDTMC for the fragment is built by applying the transformation rule 3.16 twice, then resulting the structure shown in Figure 3.24. When the property $P_{?=}(\Diamond''success'')$ is evaluated by a parametric model checker, its resulting formula is 0.999×0.999, thus equals to the reliability formula computed from its respective sequence diagram.

Finally, it is necessary to demonstrate that the reliability equivalence holds for the optional combined fragment whose semantics was changed in order to allow representing the variability of software product lines. Such an element is used twice in the sequence diagram presented in Figure 2.4b to represent the variability points `rSQlite` and `rMemory`. All other elements in such a diagram are messages (of all kinds), so its derivation tree is comprised of nodes created by applying Definitions 3.11 and 3.14, as shown in Figure 3.25. Note that the terms of the optional combined fragments will always be part of the path created by traversing the tree. Thus, the resulting formula for the reliability is $R(rOxygenation) = 0.999^6 \times rSqlite \times rMemory$, such that the variables `rSqlite` and `rMemory` assume their values when the fragment is present or 1.0 when it is absent from the configuration. Such values, indeed, can be obtained by the reliability Definition 3.14 since the exponent p varies into the set $\{0, 1\}$.

In the case of the reliability analysis by employing a parametric model checker, the transformation rules represented by Figures 3.16 and 3.19 were applied to build the FDTMC related to the `rOxygenation` fragment. The resulting FDTMC is show by Fi-

$$R(Oxygenation) = R(m_1) \qquad \boxed{m_1 = \texttt{register}}$$

$$\Big\downarrow \; (3.11)$$

$$0.999 \times R(m_2) \qquad \boxed{m_2 = \texttt{replyRegister}}$$

$$\Big\downarrow \; (3.11)$$

$$0.999 \times R(m_3) \qquad \boxed{m_3 = \texttt{sendSituation}}$$

$$\Big\downarrow \; (3.11)$$

$$0.999 \times R(m_4) \qquad \boxed{m_4 = \texttt{persist}}$$

$$\Big\downarrow \; (3.11)$$

$$0.999 \times R(opt_1) \qquad \boxed{opt_1 = \texttt{SQLite}}$$

$$\Big\downarrow \; (3.14)$$

$$\texttt{rSqlite}^{p_1} \times R(opt_2) \qquad \boxed{\begin{array}{c} opt_2 = \texttt{Memory} \\ p_1 \in 0, 1 \end{array}}$$

$$\Big\downarrow \; (3.14)$$

$$\texttt{rMemory}^{p_2} \times R(m_5) \qquad \boxed{\begin{array}{c} m_5 = \texttt{replyPersist} \\ p_2 \in 0, 1 \end{array}}$$

$$\Big\downarrow \; (3.11)$$

$$0.999 \times R(m_6) \qquad \boxed{m_6 = \texttt{replySendSituation}}$$

$$\Big\downarrow \; (3.11)$$

$$0.999$$

$$R(Oxygenation) = 0.999 \times 0.999 \times 0.999 \times 0.999 \times \texttt{rSqlite} \times \texttt{rMemory} \times 0.999 \times 0.999$$

$$R(Oxygenation) = 0.999^6 \times \texttt{rSqlite} \times \texttt{rMemory}$$

Figure 3.25: Derivation tree of reliability definitions for the *Oxygenation* and *Temperature* sequence diagrams

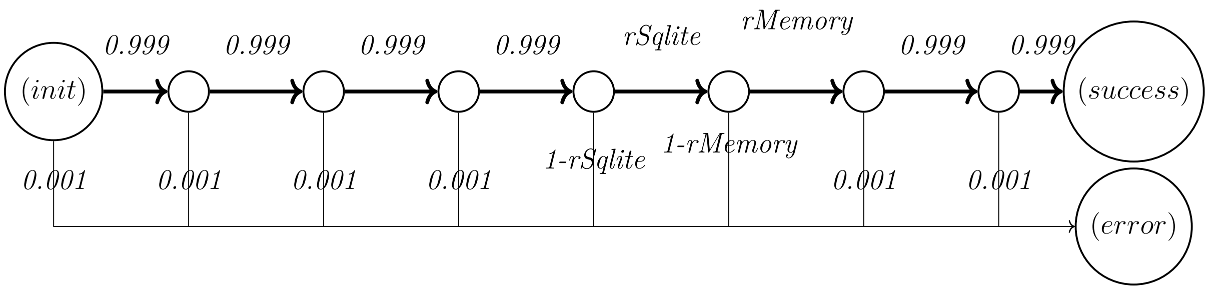


Figure 3.26: FDTMC of the *Oxygenation* and *Temperature* sequence diagrams

gure 3.26 where the unique path leading to the *success* stated is represented in bold. When the property $P_{=?}(\Diamond''success'')$ is evaluated by a parametric model checker's algorithm, it results into the formula $0.999^6 \times rSqlite \times rMemory$, where `rSqlite` and `rMemory` variables represent the reliabilities its related fragments may assume. Thus, both reliability formulae computed from the UML behavioral diagram and from the FDTMC are equal which brings evidences that reliability equivalence also holds for sequence diagrams. Such evidences are important to demonstrate that the semantics of UML sequence diagrams were preserved by their transformation rules to FDTMC sub-structures, in special, the transformation rule for the optional combined fragment (cf. Figure 3.19) that was adapted to address the variability of software product lines.

3.5 Conclusion

In short, this chapter presented how the probabilistic and variable behavior of a software product line can be represented by UML behavioral diagrams endowed with probabilities on its elements. For each behavioral element used to represent the software product line behavior a translation rule was defined in order to allow creating the FDTMC for activity or sequence diagrams. The chapter also defined the reliability notion of UML behavioral models for software product lines which, to the best of our knowledge, was not defined yet. Finally, some evidences for the reliability equivalence of both UML behavioral models and their respective FDTMCs.

Overall, the reliability function defined for UML behavioral diagrams operate over the scope of the behavioral diagram under analysis. So each element defined in a behavioral diagram will be considered as a term of the reliability function, not mattering if the element comprises another behavioral element. Such characteristic is plain when the reliability function operates over activities comprising an activity diagram and combined fragments of sequence diagrams (except the `loop`). In both cases the reliability is represented by an variable representing the reliability of its associated sequence diagram.

Chapter 4

Feature-Family-based Reliability Analysis

This chapter presents the method to evaluate the reliability property of product lines following a feature-family-based strategy [9]. It consists of three key steps, as shown in Figure 4.1.

First, the *transformation* step maps UML behavioral diagrams with variability into a graph structure called Runtime Dependency Graph (RDG), whose nodes represent the behavioral fragments and store corresponding FDTMCs (i.e. the probabilistic behavioral model), meanwhile the edges represent the runtime dependencies between such models. Next, the *feature-based* evaluation step leverages parametric model checking to analyze each FDTMC in isolation against a reliability property, by abstracting the existing runtime dependencies between them. This results in rational expressions [38] (hereafter referred to simply as *expressions*), each giving the reliability of an FDTMC as a function of the reliabilities of the FDTMCs on which it depends. Lastly, the *family-based* evaluation step follows a topological sorting of the runtime dependency graph, computing the reliability value of each configuration by evaluating the expression in each node and reusing the evaluation results previously computed for the nodes on which it depends. This step also considers the variability model of the product line in question to prune invalid configurations. The following subsections describe these steps in detail, guided by the example of Section 2.5.

4.1 Transformation

To perform the reliability analysis of a given product line, the proposed method first composes its inherent variability and probabilistic behavior into a data structure named Runtime Dependency Graph (RDG), which is then used for analysis in further steps.

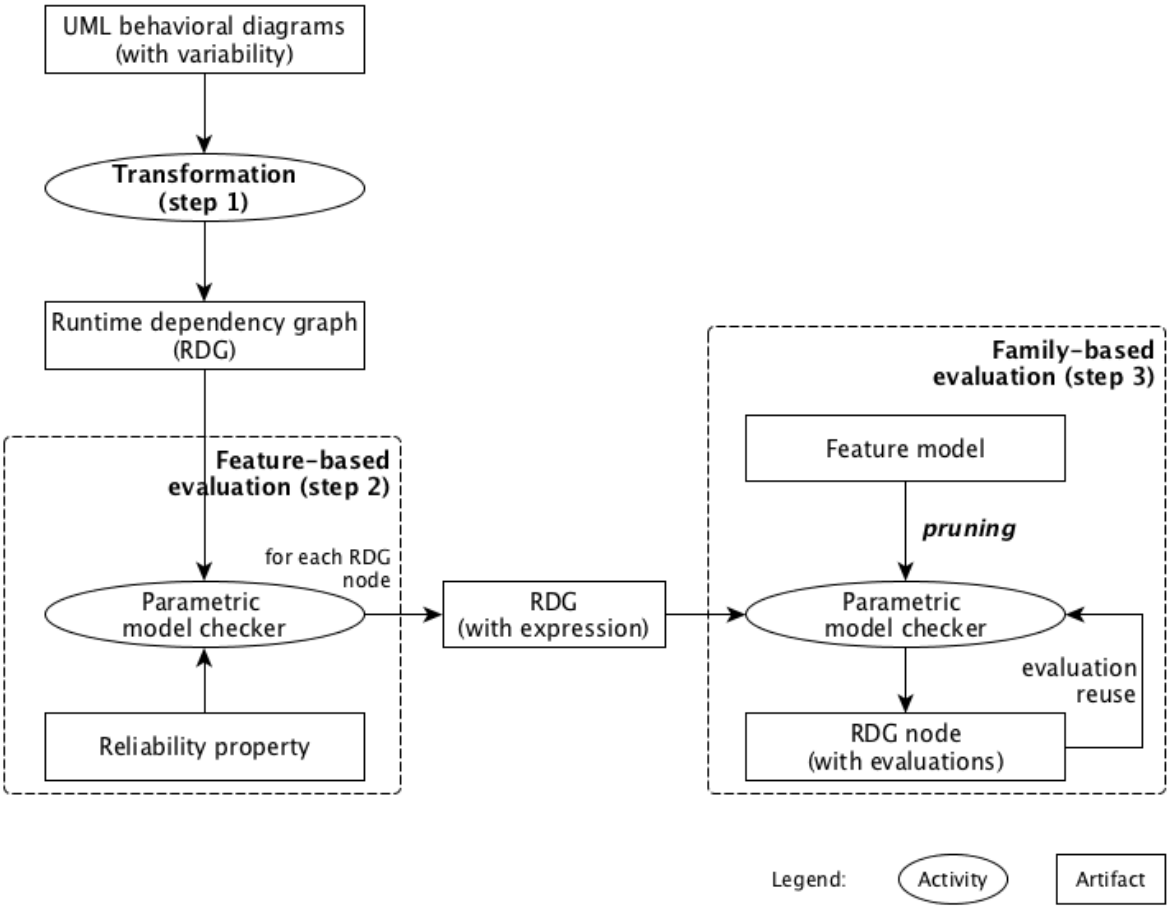


Figure 4.1: Feature-family-based approach for efficient reliability analysis of product lines

The probabilistic behavior can be derived from UML behavioral models, representing the runtime interactions between software components, enriched with reliability information for such interactions. Details on the behavioral models, the RDG, and the transformation from beahvioral models into RDG is provided next.

4.1.1 Behavioral Models

The evaluation approach that considers the coarse-grained behavior of a product line is represented by a UML activity diagram, with each activity being refined into a sequence diagram [12]. The activity diagram is useful for representing whether the activities are performed in a sequential or parallel manner, whereas sequence diagrams represent how the probabilistic behavior of the interactions between software components varies according to the configuration space of the product line. To represent probabilistic behavior, each message in a sequence diagram is annotated with a probability value that represents the reliability of the communication channel—i.e., the probability that the interaction succeeds—by using the UML MARTE profile [46] (e.g., *prob* tags in Figure 2.4b).

Without loss of generality, behavior variability is defined by *behavioral fragments*, each of which can be an activity diagram (that has an associated sequence diagram), a sequence diagram, or an optional combined fragment within a sequence diagram such that this fragment has a guard condition denoting its presence condition [44]. These conditions are propositional logical statements defined over features, that denote the set of configurations for which the guarded behavior is present. Optional combined behavioral fragments can be nested, which allows representing behavioral variability at several levels.

Note that the behavioral variability expressed by optional fragments may be implemented in two distinct ways: 1) in case the fragment's guard condition is expressed by an atomic proposition (i.e., a single feature), the feature may be implemented in its own module, which characterizes a compositional product line; 2) if the guard condition is a propositional formula comprising two or more features, such a tangled behavior can be implemented in an annotation-based style by using, for example, the `#ifdef` and `#endif` macros of the C preprocessor. Therefore, the hereby presented method can be applied to analyze both compositional and annotation-based software product lines.

As an example, Figure 2.4a shows a UML activity diagram describing, at a high level, the behavior of all products of the BSN product line. The behavior corresponding to the activity *system identifies situation* is modeled by an associated sequence diagram, *partially* depicted in Figures 2.4b and 2.4c.

The sequence diagram shown in Figure 2.4b presents three behavioral fragments whose presence conditions are the atoms `oxygenation`, `memory`, and `sqlite`. The outermost behavioral fragment represents the optional behavior for processing the oxygenation information in the BSN product line, and it varies according to two nested behavioral fragments. These latter are optional combined fragments related to *Sqlite* and *Memory* features of the feature model in Figure 2.3 and, jointly with this model's constraints, ultimately represent alternative behavior for data persistence. Analogously, the sequence diagram shown in Figure 2.4c has its behavior varying in terms of *Temperature*, *SQLite* and *Memory* features due the optional combined fragments also has the guards defined by the atoms `temperature`, `sqlite` and `memory`, respectively.

4.2 Runtime dependency graph (RDG)

A Runtime Dependency Graph (RDG) is a behavioral representation for variable systems, which combines the configurability view of a product line (expressed by presence conditions) with its probabilistic behavior (expressed by FDTMCs). Formally, it can be defined as follows.

Definition 4 [RDG] A Runtime Dependency Graph $\mathcal{R}$ is a directed acyclic graph $\mathcal{R} = (\mathcal{N}, \mathcal{E}, x_0)$, where $\mathcal{N}$ is a set of nodes, $\mathcal{E} : \mathcal{N} \cdot \mathcal{N}$ is a set of directed edges that denote a dependency relation, and $x_0 \in \mathcal{N}$ is the *root* node with in-degree 0. An RDG node $x \in \mathcal{N}$ is a pair $x = (m, p)$, where m is an FDTMC representing a probabilistic behavior and p is a propositional logic formula that represents the presence condition associated with m.

To build an RDG for a software product line, the method extracts the configurability and probabilistic information only from the UML behavioral diagrams, such that each RDG node is associated with an FDTMC derived from a behavioral fragment (c.f. Figure 3.10) and its presence condition. Since it is considered that the UML activity diagram represents the product line's *coarse-grained behavior* executed by all products and each activity is further refined (detailed) into its respective sequence diagram, two aspects of the method deccurs. First, the behavioral variability is not considered at the representation at system level, which implies its related RDG nodes have *true* as presence condition (i.e., it is satisfied for all products). Finally, as an activity has as an association with its refining sequence diagram, such association is represented in the RDG by an edge. Therefore, edges represent dependencies between nodes, which are due to refinement or nesting relations between the respective behavioral fragments. The RDG nodes that do not depend on any other node are called *basic*. The ones with dependencies are called *variant* nodes, which are represented with outgoing edges directed to the RDG nodes on which they depend.

The structure of UML sequence diagrams is tree-like, which suggests a tree could be a better model of their dependencies. Nonetheless, applications sometimes have behavioral fragments replicated throughout UML models. For instance, the data persistence behavior in Figure 2.4b is present in all fragments that denote sensor information processing including the temperature information processing represented by Figure 2.4c. In this approach, redundant fragments are represented by a single RDG node, with as many incoming edges as its number of replications. When performing this reuse, the resulting graph will be acyclic, because the original UML model is a finite hierarchy.

Figure 4.3a illustrates an excerpt of the BSN product line's RDG that represents the behavioral fragments of figures 2.4b and 2.4c. As the fragments related to *Sqlite* and *Memory* features are nested inside the fragments related to the *Oxygenation* and *Temperature* features, the RDG for these fragments represents the dependencies between their respective nodes. The behavioral fragments related to *Oxygenation* and *Temperature* are part of the sequence diagram representing the behavior of the activity *system identifies*

```
1 RDGNode transformAD(ActivityDiagram ad) {
2     RDGNode root = new RDGNode(ad.id);
3     root.model = adToFDTMC(ad);
4     root.presenceCondition = true;
5     for (Activity act : ad.activities) {
6         root.addDependency(transformSD(act.sequenceDiagram));
7     }
8     return root;
9 }
```

Listing 4.1: Activity diagram transformation

situation. Therefore, these relations are also represented by the edges from the node `rSituation` to the nodes `rOxygenation` and `rTemperature`, respectively. For brevity, it is not represented the internal structure of the nodes and the remaining RDG nodes (indicated by suspension points in Figure 4.3a).

From Behavioral Models to RDG

The transformation from behavioral models to an RDG can be described at two abstraction levels: the RDG topology and the generation of probabilistic models. Listings 4.1 and 4.2 both depict the transformation process from the topological point of view. Note that this step relies on uniquely generated identifiers for the behavioral models(line 2), which are then used as identifiers for the respective RDG nodes.

The process starts by calling the `transformAD` method (Listing 4.1), passing as argument the single activity diagram that embodies the coarse-grained behavior of the product line. This method creates the *root* node (Line 2), setting its presence condition to `true` (i.e., the overall behavior must always be present; Line 4). The root's probabilistic model is then generated by processing the input diagram with the `adToFDTMC` method (Line 3), which will be presented later. Then the approach creates an RDG node for each sequence diagram that refines an activity (denoted by the property `act.sequenceDiagram`), subsequently creating edges that mark them as dependencies of the *root* node (Line 6). Note that the *root* node is the only RDG node created by the `transformAD` method, so the root's FDTMC models the behavior represented by the activity diagram.

The creation of RDG nodes for sequence diagrams is similar: the method `transformSD` (Listing 4.2) takes a behavioral fragment as input and then creates a new RDG node whose FDTMC is derived by the `sdToFDTMC` method (Line 4). In this case, since behavioral fragments encode variability, their guard is assigned as the presence condition of the newly created node (Line 3). As with refined activities, the approach creates RDG

```
1 RDGNode transformSD(BehavioralFragment sd) {
2     RDGNode thisNode = new RDGNode(sd.id);
3     thisNode.presenceCondition = sd.guard;
4     thisNode.model = sdToFDTMC(sd);
5     for (BehavioralFragment frag : sd.optFragments) {
6         thisNode.addDependency(transformSD(frag));
7     }
8     return RDGNode.reuse(thisNode);
9 }
```

Listing 4.2: Sequence Diagram transformation

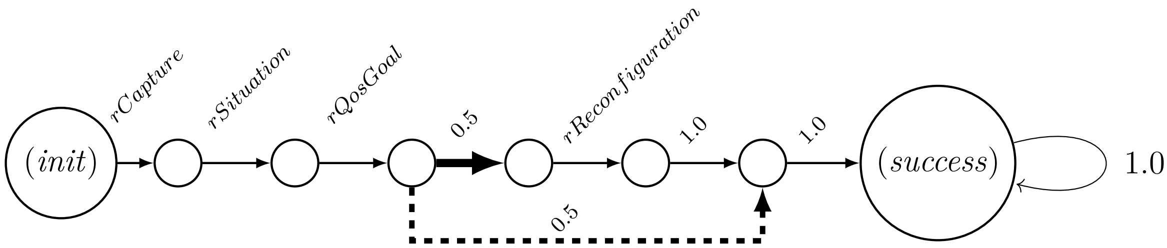
nodes for nested behavioral fragments and set them as dependencies of the node at hand (Line 6).

The reuse of behavior briefly and previously mentioned in this section is performed by calling the static method `RDGNode.reuse` (Listing 4.2, Line 8). This function maintains a registry of all RDG nodes created, and then searches among them for one that is considered *equivalent* to the one just created. This notion of equivalence is comprised of three conditions: (a) equality of presence conditions; (b) equality of FDTMCs; and (c) recursively computed equivalence of dependencies.

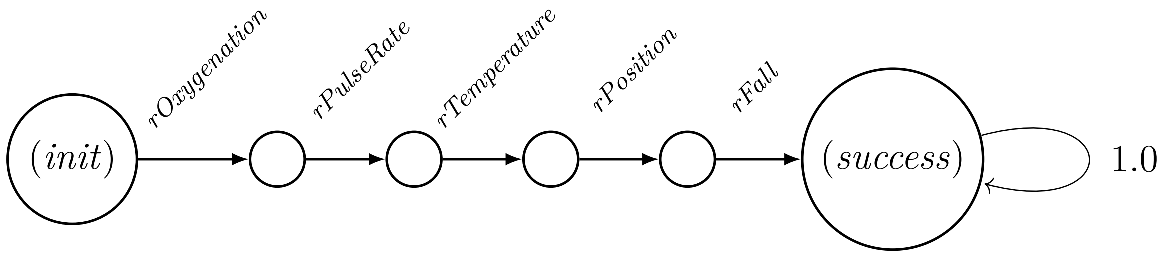
The sequence diagrams of figures 2.4b and 2.4c illustrate such result opportunity. The optional fragments related to the *SQLite* feature on both sequence diagrams fulfill the three conditions aforementioned for behavioral reuse: both fragments have the same guard condition (`SQLite`), are comprised of the same messages sequence ([`persist`,`replyPersist`]) and have the same set of dependencies (in this specific case, the $\emptyset$ set as they are basic nodes). The same rationale holds for the fragment related to the *Memory* feature. Such behavioral reuse allows reusing both models and evaluations for such fragments, fact that is represented by the incoming edges to its respective nodes in Figure 4.3a.

At the abstraction level of generating probabilistic models, the transformation of activity and sequence diagram elements into FDTMCs consists of applying the transformation templates for each considered behavioral element represented on such diagrams. These transformation templates are those addressed by Sections 3.3.1 and 3.3.2 for activity and sequence diagrams' elements, respectively.

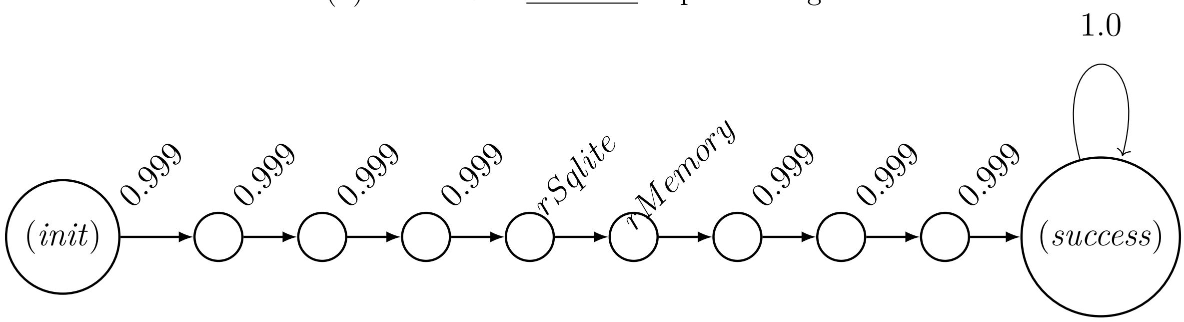
As an example, Figure 4.3a shows an excerpt of the RDG corresponding to the UML activity and sequence diagrams depicted in figures 2.4a, 2.4b and 2.4c such there is an RDG node for each kind of behavioral fragment found on all figures. Note that whenever a behavioral fragment (activity or sequence diagrams and optional combined fragment) has to be transformed, its RDG node and an edge are created to accommodate its FDTMC and represent the behavioral dependency, respectively. The node labeled `rRoot` is the root



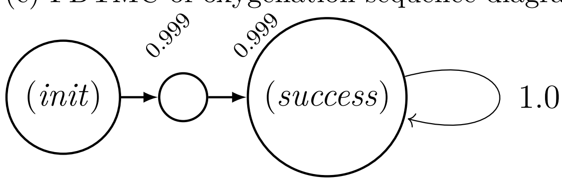
(a) FDTMC of the control loop of BSN-SPL



(b) FDTMC of situation sequence diagram



(c) FDTMC of oxygenation sequence diagram



(d) FDTMC of SQLite/Memory sequence diagram

Figure 4.2: Resulting FDTMCs

node of this RDG. The FDTMC assigned to this node (Figure 4.2a) is built by applying the transformation rules presented by Section 3.3.1 to the activity diagram in Figure 2.4a. The decision node in this activity diagram gives rise to the bold and dashed transitions in Figure 4.2a, representing the *yes* and *no* branches.

The RDG node `rSituation` represents the sequence diagrams depicted in figures 2.4b and 2.4c, corresponding to the activity *System identifies situation* of BSN's control loop (Figure 2.4a). Since this activity is performed by all products, its presence condition is *true*. The node's FDTMC, depicted in Figure 4.2b, is obtained from the sequence diagram according to the transformation templates presented by Section 3.3.2. The outgoing edges of the node `rSituation` in Figure 4.3a correspond to its dependency on the availability of sensor information—one RDG node per optional behavioral fragment. (Most of such RDG nodes corresponding to such behavioral fragments are omitted for brevity).

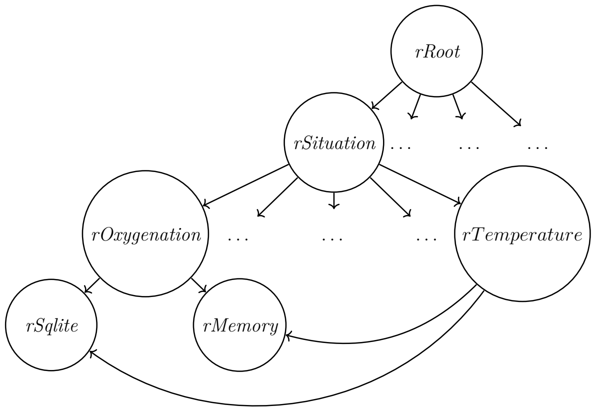
The node labeled `rOxygenation` in Figure 4.3a represents the behavior in the behavioral fragment whose presence condition is *oxygenation* (Figure 2.4b). The corresponding FDTMC is presented in Figure 4.2c. The node `rOxygenation` depends on two basic RDG nodes, `rSqlite` and `rMemory`, corresponding to the nested behavioral fragments whose presence conditions are *sqlite* and *memory*, respectively. Since both fragments have similar behavior (two sequential messages, each with reliability 0.999) their corresponding FDTMCs are equal (Figure 4.2d).

Finally, the approach relies on the divide-and-conquer strategy to decompose behavioral models. During the transformation of a behavioral fragment into an FDTMC, whenever another behavioral fragment is found, an RDG node is created with a parent-child dependency relation with the parent's RDG node. The way a software product line is decomposed results into a tree-like RDG if there is no behavioral fragment being reused. Otherwise, an RDG node representing a reused behavior fragment will have as many incoming edges as the times the fragment is reused. In this specific case, the structure of the resulting RDG will not be tree-like (that is why the RDG is a directed acyclic graph, in general).

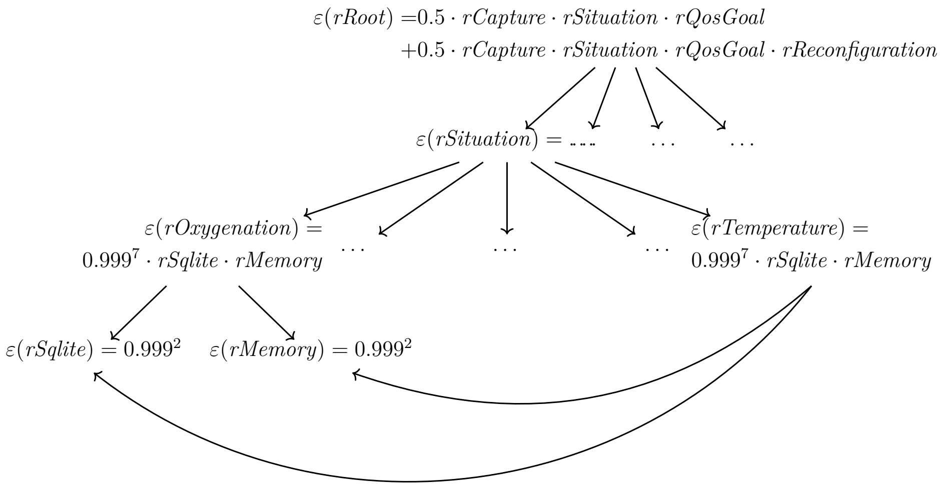
4.3 Feature-Based analysis

The role of the feature-based analysis step is to analyze the FDTMC for each RDG node in isolation, abstracting from the dependencies of other RDG nodes. That is, instead of evaluating a potentially intractable FDTMC for the product line as a whole, the approach performs this analysis as the first step of a compositional analysis by employing multiple evaluations of smaller models, one per behavioral fragment.

For each RDG node $x \in \mathcal{N}$, its FDTMC is subject to parametric model checking [47, 48]. This feature-based analysis yields x's reliability as an expression over the reliabilities of the n RDG nodes $x_1, \ldots, x_n$, on which it depends. This expression is denoted by a function $[0,1]^n \to [0,1]$, that is, the computation of a reliability value takes n reliability values as input. Therefore, there is a function $\varepsilon : \mathcal{N} \to ([0,1]^n \to [0,1])$ that yields the semantics of the reliability expression for a given RDG node. To remove possible ambiguities, the order of the formal parameters is determined by a total order relation over the corresponding RDG nodes x_i. In the case of the running example, the since the `Oxygenation` fragment depends on both `Sqlite` and `Memory` nodes such fragments are evaluated before `Oxygenation`. When analyzing RDG nodes, the same reliability



(a) RDG nodes



(b) Dependencies between expressions

Figure 4.3: RDG excerpt for the BSN product line

property of eventually reaching the *success* final state (expressed by the model checker query expression $P_{=?}[\Diamond\text{“}success\text{”}]$—see Section 2.2) is used for all FDTMCs.

Performing feature-based analysis over the RDG, as depicted in Figure 4.3a, yields the expressions shown in Figure 4.3b. These expressions illustrate that basic nodes have their reliabilities defined in terms of constants, whereas the reliabilities of variant nodes ultimately depend on the ones of basic RDG nodes. For the sake of simplicity, we overload the names of RDG nodes in Figure 4.3a as variables in the expressions in Figure 4.3b. This way, we map each variable to the RDG node whose reliability it represents.

4.4 Family-Based Analysis

The possible next step would be to evaluate the obtained expressions once for each valid configuration, so that the reliability of every product would be computed. This enumerative approach would be, in fact, a *product-based* analysis, yielding an overall *feature-product-based* analysis, similar to the one described by [11]. However, evaluating all products using this approach would be still prone to an exponential blowup, which would harm scalability.

To avoid this problem, the method hereby presented leverages a family-based analysis strategy to *lift* each expression to perform arithmetic operations over variational data, with the help of an appropriate *variational data structure* [49]. This way, it is able to represent all possible values under variation and efficiently evaluate results, sharing computations whenever possible. The data structure of choice is the Algebraic Decision Diagram (ADD)[1] [50], because it efficiently encodes a Boolean function $\mathbb{B}^n \to \mathbb{R}$ and also by the usual algebraic operations are well defined for such data structure. This way, when the variables of a formula resulting from the feature-based step is substituted by an ADD, the formula evaluation is straightforward. This is the same type as a mapping from configurations to reliability values would have, provided the Boolean values $b_1, \ldots, b_n \in \mathbb{B} = \{0, 1\}$ are taken to denote the presence (or absence) of the corresponding features $f_1, \ldots, f_n \in F$ (where F is the set of features in the feature model).

Given an expression $\varepsilon(x)$, obtained for an RDG node x in the feature-based step of the analysis (Section 4.3), the reliability ADD $\alpha(x)$ is obtained by first valuating the parameters $x_1, \ldots, x_k$ of the lifted expression with the ADDs for the reliabilities $\alpha(x_1), \ldots, \alpha(x_k)$ of the corresponding nodes upon which x depends. Then, arithmetic operations are performed using ADD semantics: for ADDs A_1 and A_2 over k Boolean variables and a binary operation $\odot \in \{+, -, \cdot, \div\}$, $(A_1 \odot A_2)(b_1, \ldots, b_k) = A_1(b_1, \ldots, b_k) \odot A_2(b_1, \ldots, b_k)$.

However, the computation of $\alpha(x)$ must take presence conditions into account. To accomplish this, the method constrains the valuation of a variable x_i with an ADD $p_x : [\![FM]\!] \to \mathbb{B}$ encoding its presence condition, with x ranging over x_1 to x_n, such n is the number of features. This ADD has the property that all configurations $c \in [\![FM]\!]$ that satisfy x_i's presence condition evaluate to 1, while all others evaluate to 0. The resulting constrained decision diagram φ_{x_i} is given by:

[1](MTBDD), generalize Binary Decision Diagrams (BDD) to Real-valued Boolean functions.

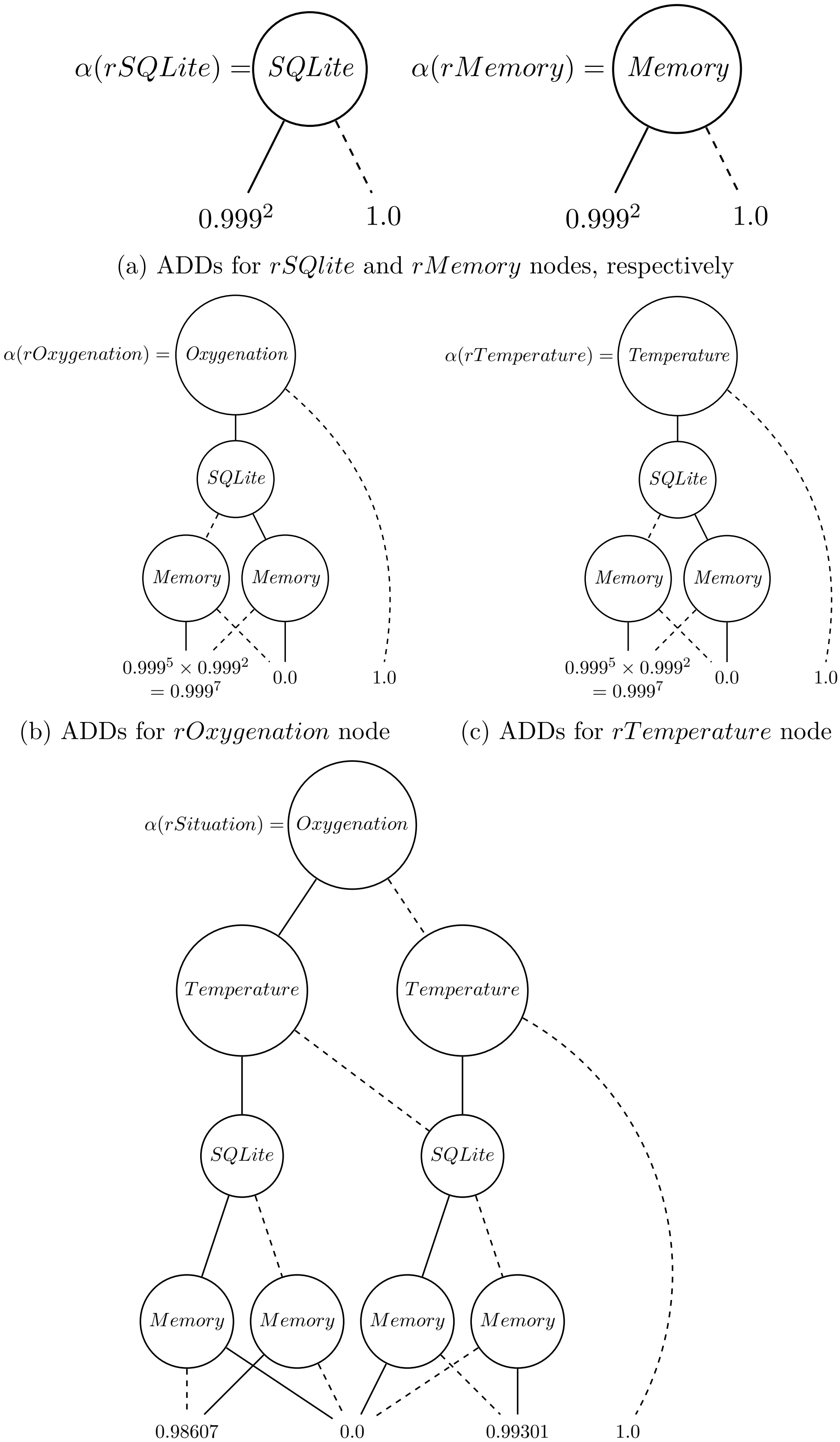
$$\varphi_{x_i}(c) = \begin{cases} \alpha(x_i)(c) & \text{if}\, p_{x_i}(c) = 1 \\ 1 & \text{otherwise} \end{cases}$$

Notice the attribution of 1 to the reliability of a behavior that is absent in a given configuration. The intuition is that, for those configurations that do not satisfy the fragment's guard conditions (i.e., $p_{x_i}(c) = 0$), the behavior represented by the optional fragment will not be part of the resulting product's behavior. Since an absent behavioral fragment has no influence on the reliability of the overall system, in practice it can be assumed 1.0 as its reliability value (i.e., it cannot fail). The ADD φ_{x_i} is obtained by means of the *if-then-else* operator for decision diagrams, and the operational details of this construction are presented in Section 5.1.

This method of evaluating the expressions is inherently recursive, since the resulting value of computing the expression for a given RDG node depends on the results of computing the expressions for the nodes on which it depends. For example, Figure 4.3b shows that the expression ε(`rOxygenation`) is defined in terms of the variables `rSqlite` and `rMemory`. Thus, before computing the lifted counterpart of expression ε(`rOxygenation`), it is necessary to compute the lifted counterparts of expressions ε(`rSqlite`) and ε(`rMemory`). The same rationale holds for the `rTemperature` node, however the α(`rSqlite`) and α(`rMemory`) are already computed and thus they can be reused. In a brief, the family-based step computes the reliabilities values each RDG node may assume by solving its ε expression using reliabilities values encoded by α for the nodes it depends on. Thus, it follows that the reliability of the product line as a whole is given by the ADD resulting from the computation of α(`rRoot`), where `rRoot` is the root RDG node.

Naturally, basic nodes are the base case of this recursion, since, by definition, they depend on no other node. Figure 4.4a depicts the ADDs representing the reliability enconding of the RDG nodes `rSqlite` and `rMemory`, respectively. Each ADD node represents a feature whose continuous edge denotes the feature's presence at the configuration, meanwhile the dashed edge means the feature is absent. Thus, α(`rSqlite`) encodes the RDG node assumes the reliability value 0.999^2 when the feature *Sqlite* is part of the configuration, otherwise 1.0. In an analogous way, the ADD for the `rMemory` RDG node represent its reliabilities values.

Figure 4.4b shows the reliability encoding computed for the `rOxygenation` RDG node. Since $\varepsilon(rOxygenation)$ is defined in terms of the variables representing the reliabilities of the nodes which it depends, $\alpha(rOxygenation)$ is computed by assigning the ADDs previously computed to `rSqlite` and `rMemory` to its respective variables in ε(`rOxygenation`),



(a) ADDs for $rSQlite$ and $rMemory$ nodes, respectively

(b) ADDs for $rOxygenation$ node

(c) ADDs for $rTemperature$ node

(d) ADDs for $rSituation$ node

Figure 4.4: ADDs for the running example

which is solved by employing the ADD's arithmetic. The resulting ADD is constrained to represent only the reliabilities of valid configurations when it is multiplied by the ADD representing the feature model's rules. In fact, all paths leading to non-zero terminal represents a valid configuration. In the case the feature *Oxygenation* is absent, its influence on the configuration's reliability is null, thus $\alpha(rOxygenation)$ assumes 1.0. Otherwise, for configurations containing *Oxygenation* and only one persistence feature (*SQLite* or *Memory*), its respective path in the ADD leads to the reliability value 0.999^7. Finally, the paths leading to the reliability value 0 represent an ill-formed configuration. For example, since *SQLite* and *Memory* are alternative features, the paths representing that both features are present or absent will lead to 0. The same rationale holds for the `rTemperature` node whose ADD is represented by Figure 4.4c. All these cases are also represented by the Table 4.1. Note that when the feature *Oxygenation* is absent for α(`rOxygenation`) (or the feature *Temperature* is absent for α(`rTemperature`)), the presence or absence of *SQLite* and *Memory* are not considered. Such effect is expected because, intuitively, since the fragment they are comprised is not part of the configuration, the presence of such features have no effect at the reliability computation.

Finally, in the case of the `rSituation` node, the ε(`rSituation`) is a formula expressed in terms of its dependent nodes which `rOxygenation` and `rTemperature` comprise. Thus, to solve $\alpha(rSituation)$ it is necessary assign the ADDs computed for `rOxygenation` and `rTemperature` to its respective variables. The resulting ADD is obtained by employing the ADD's arithmetic and it is represented by the Figure 4.4d. Such diagram is also constrained by the rules of the feature model expressed in its own ADD. Thus, all paths leading to non-zero terminals are valid partial configurations, otherwise, paths leading to the zero terminal represent ill-formed products. All these possible cases are also represented by the Table 4.1.

4.5 Conclusion

This chapter presented the evaluation method proposed for the reliability analysis of software product lines. Such method is comprised of three steps that initially, as shown by Listings4.1 and 4.2, is responsible for creating a runtime dependency graph from the behavioral models. Such step (ie. the transformation step) creates a node in the RDG for each behavioral fragment it founds while parsing the UML behavioral models.

Meanwhile the transformation step, each RDG node receives its FDTMC created according to the transformation rules afore presented. Such transformations considers the behavioral elements in the scope of the behavioral fragment in such a manner the resulting FDTMC is comprised of constants and variables that abstract the nodes which it depends

Table 4.1: Reliability of `rOxygenation`, `rTemperature` and `rSituation` fragments

Configuration (c)	α(`rOxygenation`)(c)
{Oxygenation, Sqlite, ¬Memory}	995*(998/1000)*1/1000 = 0,99301
{Oxygenation, ¬Sqlite, Memory}	995*1*(998/1000)/1000 = 0,99301
{¬Oxygenation, ____, ____}	1,0
{Oxygenation, Sqlite, Memory}	–
{Oxygenation, ¬Sqlite, ¬Memory}	–
Configuration (c)	α(`rTemperature`)(c)
{Temperature, Sqlite, ¬Memory}	995*(998/1000)*1/1000 = 0,99301
{Temperature, ¬Sqlite, Memory}	995*1*(998/1000)/1000 = 0,99301
{¬Temperature, ____, ____}	1,0
{Temperature, Sqlite, Memory}	–
{Temperature, ¬Sqlite, ¬Memory}	–
Configuration (c)	α(`rSituation`)(c)
{Oxygenation, ¬Temperature, Sqlite, ¬Memory}	(99301/10000)*1 = 0,99301
{Oxygenation, ¬Temperature, ¬Sqlite, Memory}	(99301/10000)*1 = 0,99301
{Oxygenation, ¬Temperature, Sqlite, Memory}	–
{Oxygenation, ¬Temperature, ¬Sqlite, ¬Memory}	–
{¬Oxygenation, Temperature, Sqlite, ¬Memory}	1*(99301/10000) = 0,99301
{¬Oxygenation, Temperature, ¬Sqlite, Memory}	1*(99301/10000) = 0,99301
{¬Oxygenation, Temperature, Sqlite, Memory}	–
{¬Oxygenation, Temperature, ¬Sqlite, ¬Memory}	–
{Oxygenation, Temperature, Sqlite, ¬Memory}	(99301/10000)*(99301/10000) = 0,98607
{Oxygenation, Temperature, ¬Sqlite, Memory}	(99301/10000)*(99301/10000) = 0,98607
{Oxygenation, Temperature, Sqlite, Memory}	–
{Oxygenation, Temperature, ¬Sqlite, ¬Memory}	–
{¬Oxygenation, ¬Temperature, ____, ____}	1,0*1,0 = 1,0
{¬Oxygenation, ¬Temperature, ____, ____}	1,0*1,0 = 1,0

on. Then, the feature-based step employs the parametrical model checker's algorithm in order to obtain the formula ε representing the reliability of each node. Thus, this jointly use of transformation and feature-based steps walk through the behavioral modeling in a top-down fashion, in order to discover and transform the behavioral fragments as soon as their dependencies are revealed.

When the RDG structure is finished, the family-based step takes place and runs back all the runtime dependency graph in order to solve the reliability expressions ε taking into account the presence conditions associated to each RDG node and the feature model's rules. The result is represented in the suitable variational data structure named ADD such that, for each RDG node, all valid partial configurations and their reliabilities are represented by means of the α function (such function is represented by the ADD). Thus, the family-based step follows a bottom-up strategy to solve all the expressions stored in RDG nodes, in such a manner that the $\alpha(root)$ stores the reliabilities of all full and valid configurations of the software product line.

Chapter 5

Proposal Evaluation

To assess the merits of the feature-family-based strategy, initially the key aspects of its implementation (Section 5.1) are highlighted, followed by its complexity analysis (Section 5.2). Finally an empirical evaluation (Section 5.3), the threats to its validity (Section 5.4) are presented.

5.1 Implementation

The evaluation method presented hereby is implemented as a new tool named ReAna (**Re**liability **Ana**lysis), whose source code is open and publicly available[1]. ReAna takes as input a UML behavioral model, for example, built using the MagicDraw tool[2], and a feature model described in conjunctive normal form (CNF), for example, as exported by FeatureIDE [51]. It then outputs the ADD representing the reliability of all products of the product line to a file in DOT format, and it prints a list of configurations and respective reliabilities. The latter can be suppressed or filtered to a subset of possible configurations of interest.

ReAna uses PARAM 2.3 [47] to perform parametric model checking and the CUDD 2.5.1 library[3] for ADD manipulation. However, any other tool or library providing the same functionality (e.g., the parametric model checker from [48]) could be used, too.

ReAna's main evaluation routine is depicted in Listing 5.1. After parsing and transforming the input models into an RDG structure (see Section 4.1), the method `evalReliability` is invoked on the RDG's root node. Its first task is to perform a topological sort of the RDG nodes, so that it obtains a list in which every node comes after all the nodes on which

[1]https://github.com/SPLMC/reana-spl
[2]http://www.nomagic.com/products/magicdraw.html
[3]ftp://vlsi.colorado.edu/pub/cudd-2.5.1.tar.gz

```
1 ADD evalReliability(RDGNode root) {
2     List<RDGNode> deps = root.topoSortTransitiveDeps();
3     LinkedHashMap<RDGNode, String> expressionsByNode =
          getReliabilityExpressions(deps);
4     Map<RDGNode, ADD> reliabilities = evalReliabilities(expressionsByNode);
5     return reliabilities.get(root);
6 }
```

Listing 5.1: ReAna's main evaluation routine

```
1  ADD evalNodeReliability(RDGNode node,
2                          String reliabilityExpression,
3                          Map<RDGNode, ADD> relCache) {
4      Map<String, ADD> depsReliabilities = new HashMap();
5      for (RDGNode dep: node.getDependencies()) {
6          ADD depReliability = relCache.get(dep);
7          ADD presCond = dep.getPresenceCondition();
8          ADD phi = presCond.ifThenElse(depReliability,
9                                        constantAdd(1));
10         depsReliabilities.put(dep.getId(), phi);
11     }
12     ADD reliability = solve(reliabilityExpression,
13                             depsReliabilities);
14     return FM.times(reliability);
15 }
```

Listing 5.2: Evaluation of the reliability function for a single node

it (transitively) depends (Line 2). This implements the recursion described in Section 4.4 in an iterative fashion.

Then, it proceeds to parametric model checking of the reliability property in the FDTMC corresponding to each of the nodes (Line 3). Although this step does not depend on the ordering of nodes (because it handles dependencies as variables), it is useful that its output respects this order. This way, the resulting reliability expressions (ε in Section 4.3) can be evaluated in an order that allows every variable to be immediately resolved to a previously computed value, thus eliminating the need for recursion and null checking.

The third step is to evaluate each reliability expression, which yields an ADD representing the reliability function (α in Section 4.4) for each of the nodes. The evaluation of such reliability ADDs (method `evalReliabilities` in Line 4, Listing 5.1) invokes, for each node, method `evalNodeReliability`, which we present in Listing 5.2. It computes the φ functions of a node's dependencies (as in Section 4.4), encoding satisfaction of their presence conditions by means of conditionals in ADD `ITE` (*if-then-else*) operations (Line 8, Listing 5.2). The reliability function of each dependency is looked up in a reliability cache (`relCache`, in Line 6, Listing 5.2) and is then used as the *consequent* argument of the `ITE` operator, with the *alternative* argument being the constant ADD corresponding to 1.

After all these functions are computed, they are used to evaluate the lifted reliability expression (Line 12, Listing 5.2). Whenever a variable appears in this expression, function φ of the corresponding RDG node (on which the current one depends) is looked up in a variable–value mapping, indexed by the node id (`depsReliabilities`).

When this evaluation of α is done, it is necessary to consider only the valid configurations for the node at hand by discarding the reliability values of ill-formed products. The feature model's rules are represented by an ADD where all paths leading to terminal 1 represent a valid configuration, otherwise the path leads to terminal 0. Thus, for the node under evaluation the invalid configurations are pruned by multiplying its reliability ADD by the one representing the feature-model's rules (Line 14, Listing 5.2), so the resulting ADD yields the value 0, for ill-formed products and the actual reliability for the valid ones.

All reliabilities computed in this way are progressively added to the reliability cache *relCache*. At the end of this loop inside `evalReliabilities`, the cache contains the reliability function for every node and is then returned (Line 4, Listing 5.1). The reliability of interest is then the one of the root RDG node (the one argument to `evalReliability`, Listing 5.1), so it is queried in constant time because of the underlying data structure.

5.2 Analytical Complexity

The overall analysis time is the sum of the time taken by each of the sequential steps in Listing 5.1. First, the computation of an ordering that respects the transitive closure of the dependency relation in an RDG (Line 2) is an instance of the classical topological sorting problem for directed acyclic graphs, which is linear in the sum of nodes and edges [52].

Second, the computation of the reliability expression for an RDG node consists of a call to the PARAM parametric model checker, which requires n calls to cover all nodes (Line 3). The parametric model checking problem for a model of s states consists of $O(s^3)$ operations over polynomials, each of which depends on the number of monomials in each operand [38]. This number of monomials is, in the worst case, exponential in the number of existing variables. The number of variables for a given node is, in turn, dependent on its number of child nodes and on the modeled behavior (e.g., if there are loops or alternative paths). Thus, the time complexity of computing all the reliability expressions is linear in the number of RDG nodes, but depends on the topologies of the RDG and of the models represented by each of its nodes (such dependencies are addressed with more details later on).

Last, method `evalReliabilities` calls method `evalNodeReliability`, which corresponds to the reliability function α in Section 4.4, once for each node. `evalNodeReliability`'s complexity is dominated by that of ADD operations, which are polynomial in the size of the operands [50]. Indeed, for ADDs f, g, and h, the *if-then-else* operation $\texttt{ITE}(f, g, h)$ is $O(|f| \cdot |g| \cdot |h|)$. Likewise, $\texttt{APPLY}(f, g, \odot)$, where $\odot$ is a binary ADD operator (e.g., multiplication), is $O(|f| \cdot |g|)$. Here, $|f|$ denotes the size of the ADD f, that is, its number of nodes. Because of configuration pruning (Section 4.4), all ADD sizes in our approach are bound by $|FM_{ADD}|$ (i.e., the size of the ADD that encodes the rules in the feature model).

Since the evaluation of α for a given node comprises a number of operations on the reliability ADDs of the nodes on which it depends (Listing 5.2, Line 12), an upper bound estimate for polynomial arithmetics must be provide. If a node identified by x has c children (nodes on which it depends), $\varepsilon(x)$ is a polynomial in c variables and it has, at most, e_{max}^c monomials of c variables each, where e_{max} is the maximum exponent for any variable. Each monomial has in turn, at most, $2c$ operations: c exponentiations and c multiplications among variables and the coefficient. Also, no variable can have an exponent greater than the maximum number of transitions between the initial and the success states of the original FDTMC, and this number is itself bound by the number m of messages in the corresponding behavioral model fragment. Thus, the number of ADD operations needed to compute this reliability ADD is $O(c \cdot m^c)$. This leads to an evaluation time of $O(c \cdot m^c \cdot |FM_{ADD}|^2)$.

Since the reliability of each RDG node needs to be evaluated exactly once (due to caching), there are n computations of $\alpha(x_i)$, one for each of the n RDG nodes x_i. Hence, the cumulative time spent on reliability functions computation is $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot |FM_{ADD}|^2)$, where c_{max} is the maximum number of children per node, and m_{max} is the maximum number of messages per model fragment.

Although this complexity bound is quadratic in the number of features, the number of nodes in an ADD is, in the worst case, exponential in the number of variables. As the variables in FM_{ADD} represent features, this means $|FM_{ADD}|$ can be exponential in the number F of features. Hence, the worst-case complexity is $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot 2^{2 \cdot F})$. This worst-case exponential blowup cannot be avoided theoretically, but, in practice, efficient heuristics can be applied for defining an ordering of variables that can cause the ADD's size to grow linearly or polynomially, depending on the functions being represented [10]. Thus, as the growth in the sizes of ADDs varies with the product line being analyzed [53] and is, at least, linear in the number of features, it can also state the best-case time complexity is $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot F^2)$.

In summary, the time complexity of the feature-family-based analysis strategy lies between $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot F^2)$ and $O(n \cdot c_{max} \cdot m_{max}^{c_{max}} \cdot 2^{2 \cdot F})$, where n is the number of RDG nodes, c_{max} is the maximum number of child nodes in an RDG node, m_{max} is the maximum number of messages in a behavioral fragment, and F is the number of features of the product line.

5.3 Empirical Evaluation

The empirical evaluation aims at comparing the feature-family-based analysis strategy (c.f. Chapter 4) with other state-of-the-art strategies for product-line reliability analysis, as identified by [9]: product-based, family-based, feature-product-based, and family-product-based. It is expected that the feature-family-based approach performs better than the others, since it (a) decomposes behavioral models into smaller ones and (b) prevents an exponential blowup by computing the reliabilities of all products at once using ADDs. The comparison focuses on the practical complexity of the selected strategies and is guided by the following research question:

- **RQ1:** How do product-line reliability analysis strategies compare to one another in terms of time and space?

To address RQ1, it was measured the time and space demanded by each strategy for the analysis of six available software product lines and augmented versions thereof. For the time measure, the wall-clock time spent during analysis after model transformation was considered, including the recording of reliability values for all configurations of a given product line. Transformation time was excluded from this measurement, because all the implementations of the analysis strategies employ the same transformation routines (using the rules presented in Section 4.2). From the transformation step on, the analysis strategies start to differ as each one traverses the resulting FDTMC in its specific fashion. For the space measure, the peak memory usage for each strategy during the evaluation of each product line was considered. This empirical assessment is described in detail in the following subsections.

5.3.1 Subject Systems and Experiment Design

To empirically compare the complexity of the different analysis strategies, the experiment started with the models of six available product lines. Table 5.1 shows the number of features, the size, and the characteristics of the solution space of each one of these product lines. The solution space is described in terms of the number of activities in the activity diagram and of the total number of behavioral fragments present in the sequence diagrams. The general criterion for choosing these systems was the availability of their variability model. EMail, MinePump, BSN, and Lift were chose due to the fact that they had been commonly used in previous work studying model checking of product lines [13, 14, 12]. InterCloud and TankWar product lines were selected due to the significant size of their configuration spaces.

Table 5.1: Initial version of product lines used for empirical evaluation

			Solution Space's Characteristics	
	# Features	**# Products**	**# Activities**	**# Behavioral fragments**
EMail [30]	10	40	4	11
MinePump [28]	11	128	7	23
BSN [12]	16	298	4	15
Lift [29]	10	512	1	10
InterCloud [31]	54	110592	5	51
TankWar [30]	144	4.21×10^{18}	7	81

Each of the six original systems was evolved 20 times, with each evolution step adding one optional feature and a corresponding behavioral fragment with random messages defining its probabilistic behavior. According to Section 4.1.1, the name of the newly introduced feature was assigned as the guard condition of each new behavioral fragment, and each message in a fragment received a probability value. Thus, each evolution step doubles the size of the configuration space of the subject product line, with an optional behavior for the added feature.

The independent variable of the experiment is the evaluation strategy employed to perform the reliability analysis. The dependent variables are the metrics for time and space complexity. Each subject system was evaluated by all treatments.

The outcomes were analyzed using statistical tests, to properly address outlying behavior and spurious results. This way, it is more likely to overrule factors that affect performance but are difficult to control (e.g., JVM warm-up time and OS process scheduling). Ideally (i.e., disregarding uncontrollable factors), it is expected all runs of a given analysis strategy over the same subject product line to yield the same result. Thus, instead of comparing isolated runs of different strategies, the inferred distribution of results of all runs of a strategy were compared to the corresponding distribution for another strategy. Since there were multiple analysis strategies to compare with, the comparison was accomplished pairwise with the feature-family strategy, for example, feature-based with feature-family-based or family-based with feature-family-based.

Standard statistical tests for equality of the pairs of samples were applied. The null hypothesis was that both samples come from the same distribution, while the alternative hypothesis was that one comes from a distribution with larger mean value than the other. The specific statistical test was the Mann-Whitney U test whenever one of the samples, at least, was not normally distributed. Otherwise, the t test for independent samples was applied in the case the variances were equal, or Welch's t test in case of different variances. The significance level for all tests was 0.01.

5.3.2 Experiment setup

Modeling We implemented each strategy as a variant of ReAna, thus relying on the same tools and libraries for model checking, ADD manipulation, and expression parsing (see Section 5.1). These ReAna extensions are also publicly available at its GitHub's repository[4]. Graduate students created the input UML behavioral models using MagicDraw 18.3 with Marte UML profile. All models were validated by the research group which the author comprises.

Instrumentation For this experiment a tool called SPL-Generator was implemented in order to create valid feature and behavioral models of a product line, according to a set of parameters (more details in Appendix B). This tool was used to create evolution scenarios, in order to assess how each evaluation strategy behaves with the growth of the configuration space. To obtain data regarding analysis time, Java's standard library method `System.nanoTime()` was used to get the time (with nanoseconds precision) reported by the Java Virtual Machine immediately before and right after ReAna's main analysis routine (Listing 5.1). The difference between these two time measures is taken to be the elapsed analysis time. Space usage was measured using the maximum resident set size reported by the Linux `/usr/bin/time` tool. This value represents the peak RAM usage throughout ReAna's execution.

Evolution Scenarios the SPL-Generator tool was used to evolve each software product line chose as a subject system of the empirical evaluation, according to the representation provided in Figure 5.1. This evolution was accomplished stepwise, and it started with the original feature model FM_0 (created by FeatureIDE) and behavioral models BM_0 (created by MagicDraw)—this set of models is hereafter refered to as *original seed* or $seed_0$. At each evolution step ev_i, the generator tool doubled the configuration space of the subject system by adding an optional feature in order to generate a new feature model FM_i(no cross-tree constraint was added, to avoid constraining configuration

[4] http://github.com/SPLMC/reana-spl

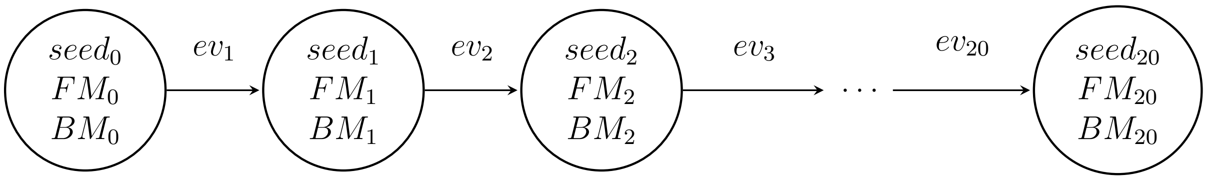


Figure 5.1: Evolution of subject systems accomplished by the SPL-Generator tool

space growth). For the newly created feature, the generator tool also creates an optional behavioral fragment comprising 10 messages randomly generated between 2 lifelines randomly chosen from a set of 10 lifelines. To establish a relation between the new feature and the corresponding new behavioral fragment, the fragment's guard condition is defined as being the atomic proposition containing the new feature's name, which characterizes the evolutions as being compositional. However, it is worth mentioning that the evaluation method also applies to the analysis of annotation-based software product lines since it was able to evaluate the original version of the EMail subject system ($seed_0$ that contains optional fragments expressed by a conjunction of two features thus, following an annotation-based implementation) and its evolutions. Each lifeline received a random reliability value from the range $[0.999, 0.99999]$. The guard condition of the behavioral fragment received an atomic proposition named after the feature, to relate the newly created items. The *topological allocation* method was used by the generator tool to create the new behavioral model BM_i, so the nesting of sequence diagrams follows the feature relations in the feature model. The end of an evolution step results into a new version of the product line ($seed_i$), which will be considered as a new *seed* for the next evolution step. Each subject system was evolved 20 times, as shown in Figure 5.1, and all artifacts are available at the supplementary site[5] created for a paper's submission.

Measurement Setup The experiment was executed by using twelve Intel i5-4570TE, 2.70GHz, 4 hyper-threaded cores, 8 GB RAM and 1 GB swap space, running 64-bit CentOs Linux 7. The experiment environment (i.e., the set of tools, product line models, and automation running scripts) was defined as a Docker[6] container[7] running 64-bit Ubuntu Linux 16.10, with access to 4 cores and 6 GB of main memory of the host machine. Each subject system was evaluated 8 times by each analysis strategy in each machine, thus summing up 96 evaluations for each pair of subject system and strategy. Because of the number of evaluations, it was defined a limit of 60 minutes for analysis execution time, after which the analysis at hand would be canceled. The results were then grouped to

[5] `https://splmc.github.io/scalabilityAnalysis/`
[6] https://www.docker.com/
[7] `https://hub.docker.com/r/andrelanna/reana-spl/`

perform the time and memory consumption analysis. The evaluations that exceeded the time limit were discarded from the statistical analysis.

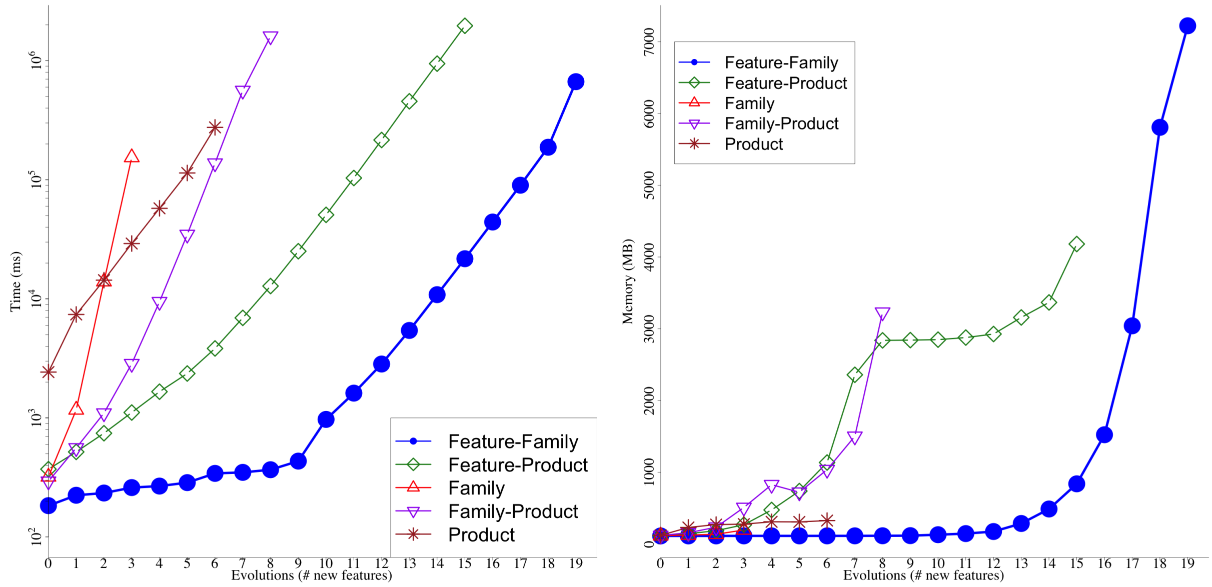
5.3.3 Results and analysis

Figures 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7 show plots with the mean time and memory demanded to analyze the Email, MinePump, BSN, Lift, InterCloud, and TankWar product lines (and corresponding evolutions), respectively. The horizontal axes represent the number of added features (with respect to the original product line) in the analyzed models. Thus, they range from 0 (the original model) to 20 (last evolution step). The vertical axes represent either the time in milliseconds (in logarithmic scale) or the space in megabytes.

The values of the plots are available in Tables 1 and 2 of Appendix A. Statistical tests over both time and space data rejected the null hypothesis for all pairs of strategies. Thus, within a significance level of 0.01, we can assume no two samples come from distributions with equal means.

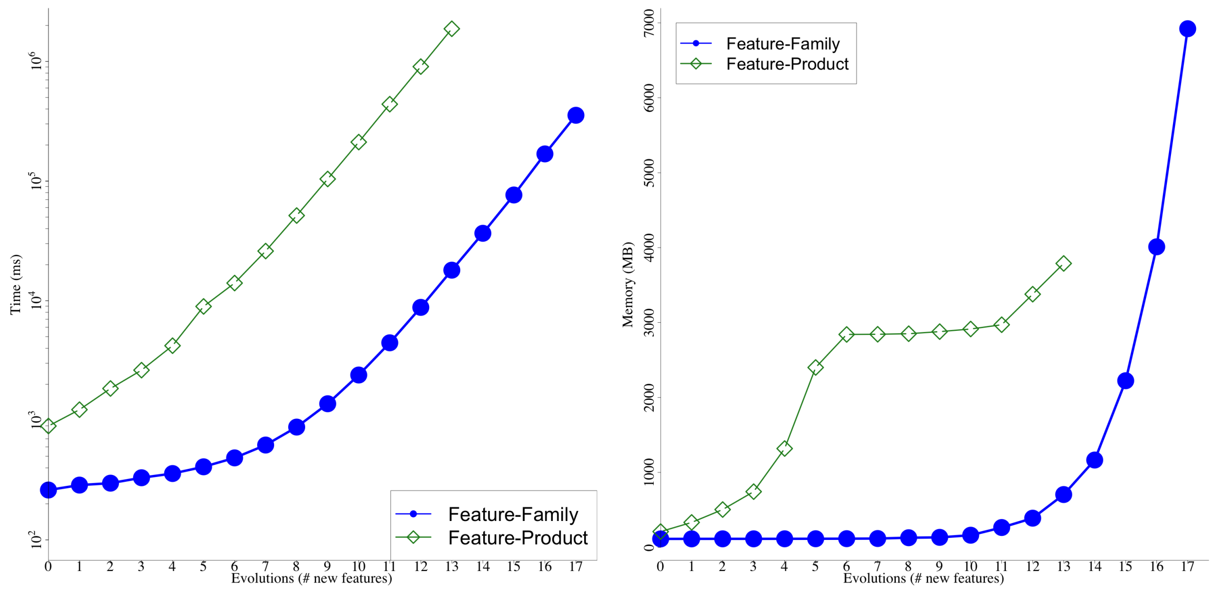
Overall, the experiments show with statistical significance that the feature-family-based strategy is faster than all other analysis strategies (as shown in Figures 5.2a, 5.3a, 5.4a, 5.5a, 5.6a, and 5.7a). Regarding execution time, in the worst case, the feature-family-based strategy performed 60% faster than the family-product-based strategy, when analyzing the original models of the Email product line (Figure 5.2a); in the best case, it outperformed the family-product-based analysis of the BSN product line with 4 optional features added (i.e., its 5^{th} evolution step—Figure 5.4a) by 4 orders of magnitude. Such cases are highlighted in yellow in Table A1. Regarding memory consumption (Figures 5.2b, 5.3b, 5.4b, 5.5b, 5.6b, and 5.7b), the experiment also shows with statistical significance that, in the worst case, the feature-family-based strategy demanded 2% less memory than the family-based strategy when analyzing the original model of the Lift product line; in the best case, it saved around 4,757 megabytes when analyzing the 3^{rd} evolution step of the InterCloud product line. Such cases are highlighted in yellow in Table A2.

The feature-family-based strategy also scaled better in response to configuration space growth in comparison with other strategies. In the worst case, this strategy scaled up to a configuration space one order of magnitude larger than the limit of the nearest scalable strategy (the feature-product-based analysis of the Email, MinePump, BSN, and Lift systems). In the best case, the feature-family-based strategy supported a configuration space 5 orders of magnitude larger than supported by the feature-product-based strategy (when analyzing the InterCloud product line). Finally, it worths highlighting that only feature-family-based strategy was able to analyze the TankWar product line, from its



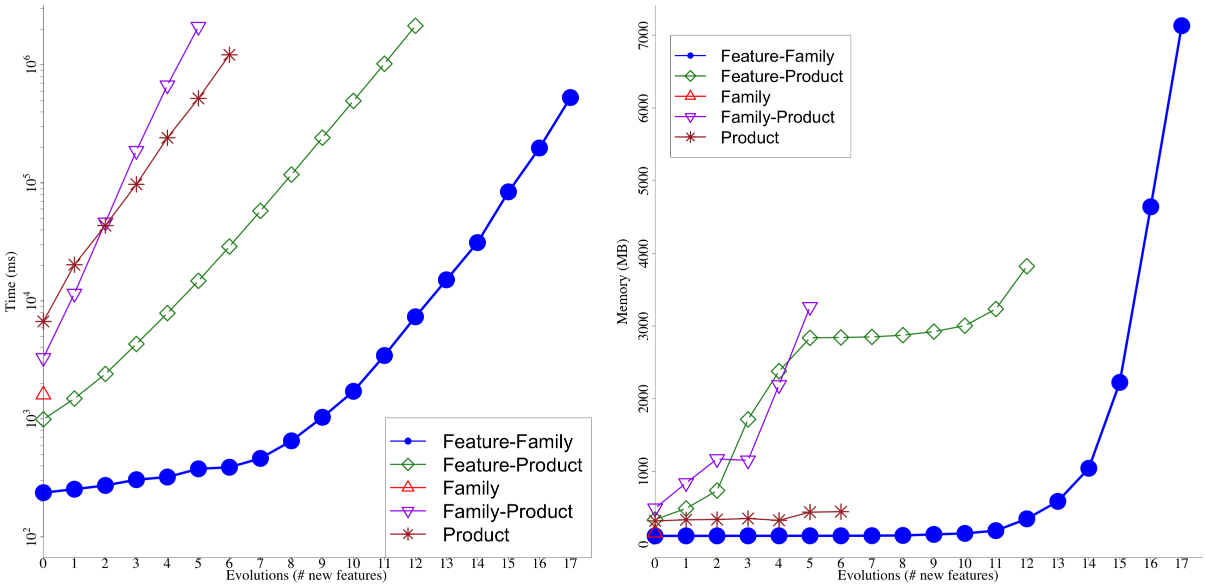
(a) Analysis time (b) Demanded memory

Figure 5.2: Time and memory required by different analysis strategies when evaluating evolutions of Email System



(a) Analysis time (b) Demanded memory

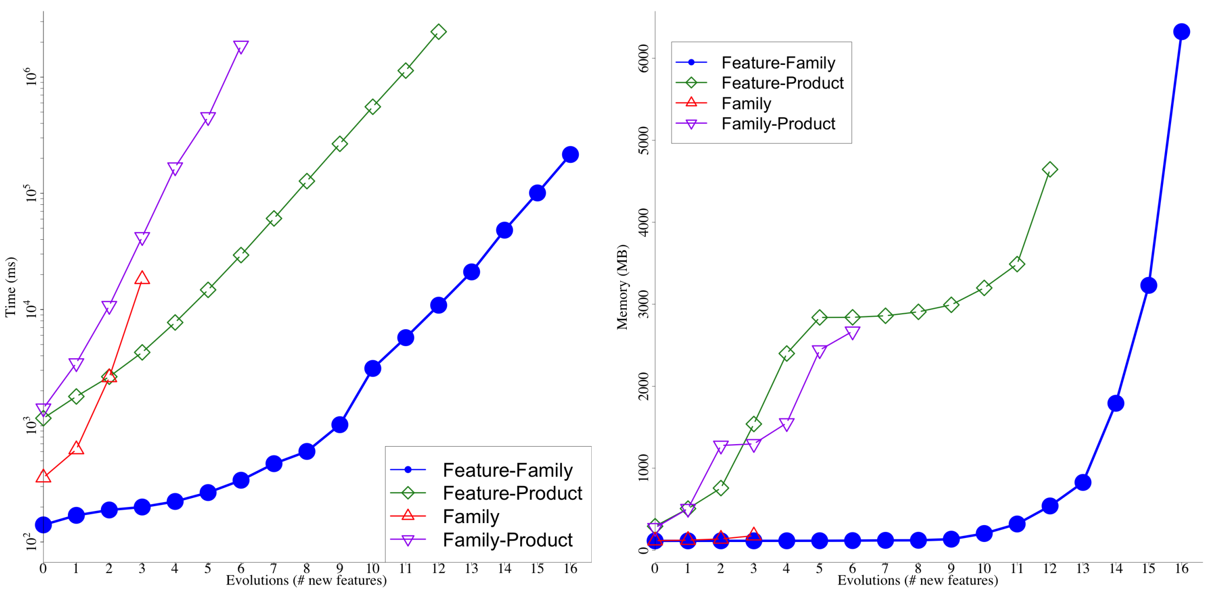
Figure 5.3: Time and memory required by different analysis strategies when evaluating evolutions of MinePump System



(a) Analysis time

(b) Demanded memory

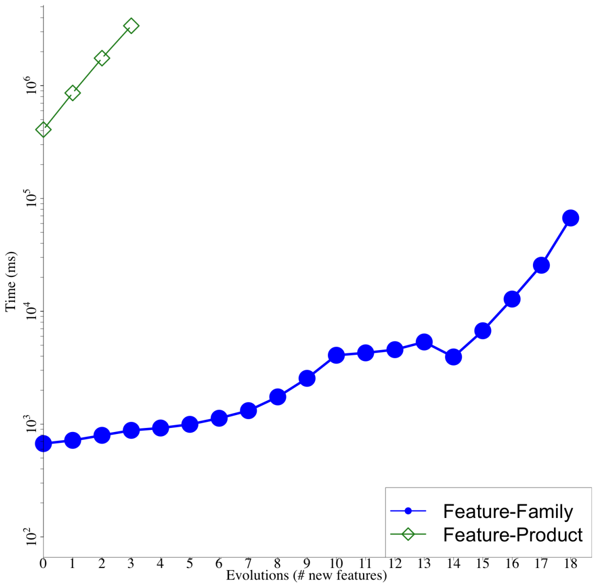
Figure 5.4: Time and memory required by different analysis strategies when evaluating evolutions of BSN-SPL



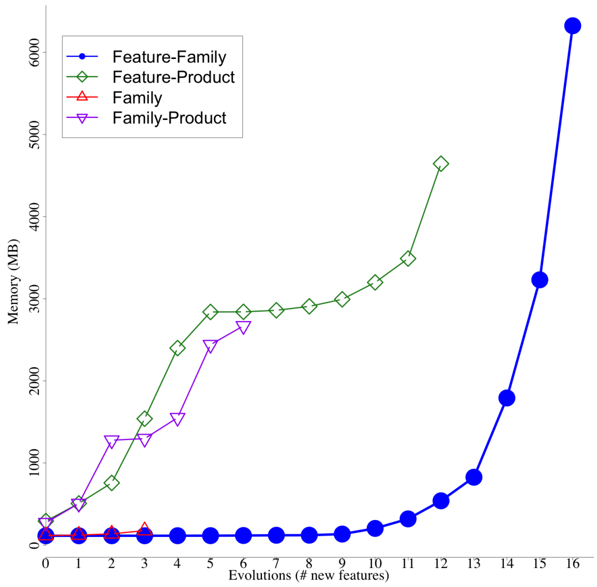
(a) Analysis time

(b) Demanded memory

Figure 5.5: Time and memory required by different analysis strategies when evaluating evolutions of Lift System

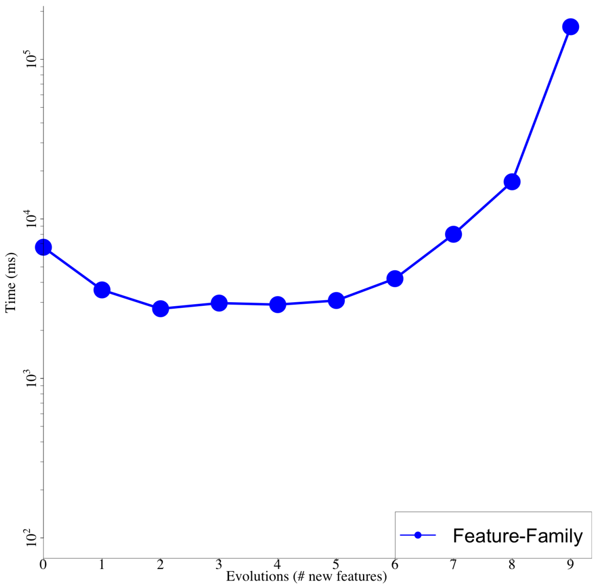


(a) Analysis time

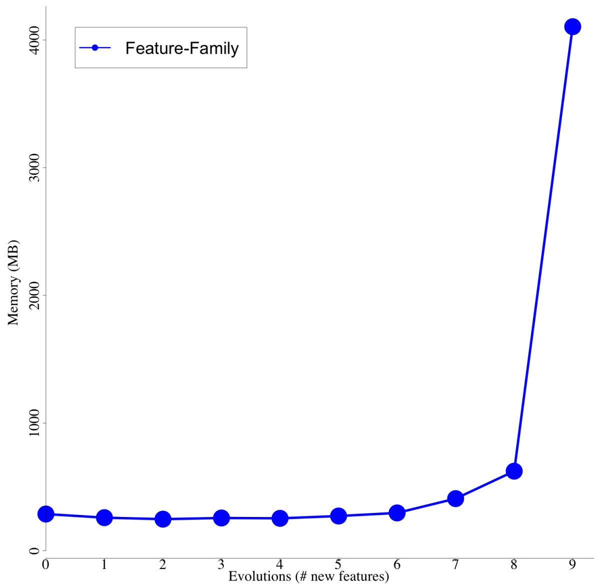


(b) Demanded memory

Figure 5.6: Time and memory required by different analysis strategies when evaluating evolutions of InterCloud System



(a) Analysis time



(b) Demanded memory

Figure 5.7: Time and memory required by different analysis strategies when evaluating evolutions of TankWar battle game

original model up to its 9^{th} evolution step. That is, the feature-family-based strategy was able to analyze the reliability of up to 10^{21} products within 60 minutes.

Table 5.2: Probabilistic models statistics

SPL	Feature-*			Family-*		Product	
	# states	# variables	# models	# states	# variables	# states	# models
EMail	12	0.93	14	182	9	115.8	40
MinePump	7.26	0.95	23	289	10	155.5	128
BSN	11.37	1.44	16	238	12	136.56	298
Lift	12.91	0.91	11	153	10	114	512
InterCloud	7.4	0.98	52	437	47	352.25	110592
TankWar	8.30	0.99	79	735	69	≈500	4.21×10^{18}

5.3.4 Discussion

One reason for the feature-family-based strategy being faster than the alternatives is that it computes the reliability values of a product line by model checking a small number of comparatively simple models. In contrast, family-based and family-product-based strategies yield more complex probabilistic models than the others, trading space for time. The complementary explanation for the performance boost is that the family-based analysis step leverages ADDs to compute reliability values, which leads to fewer operations than necessary if these values were to be calculated by enumeration of all valid product line configurations (cf. Section 5.2).

Table 5.2 shows the average number of states and variables present in the models created by each analysis strategy[8], with feature-family-based and feature-product-based strategies grouped under Feature-*, and family-based and family-product-based ones grouped under Family-*. Some values are omitted, because the number of models is always 1 for family-based approaches, and the number of variables is always 0 for product-based ones. In this table, all probabilistic models created by Feature-* analyses have, indeed, fewer states than the ones generated during Family-* and product-based approaches. Feature-based models also have fewer variables than the corresponding family-based ones.

The plots of the experiment results reveal some characteristics that depart from the expected behavior, which is addressed next. First, there is a single data point for the family-based analysis of the BSN product line (Figure 5.4), despite its analysis time being in the order of seconds (far from reaching the time limit). In fact, the family-based strategy was able to analyze BSN's models up to the 6^{th} evolution step. However, the resulting expression representing the family's reliability contained numbers that exceeded Java's floating-point representation capabilities. Thus, converting these numbers to the

[8]For TankWar, the average number of states in the product-based case is an estimate, because it is impractical to generate all models.

`double` data type yielded *not a number* (`NaN`). To the best of our knowledge, the overflow of floating-point representation was not reported yet by previous studies addressing reliability analysis of software product lines.

The second remarkable characteristic are the plateaus for feature-product-based analysis at the memory plots in Figures 5.2b, 5.3b, 5.4b, and 5.5b. The hypothesis is that this behavior is related to the memory management of the Java Virtual Machine (JVM), but a detailed investigation was out of scope.

It is also evident that the plots for feature-family-based analysis are monotonically increasing, with two exceptions: a single decrease at the 14^{th} evolution step of the Intercloud product line (Figure 5.6) and a "valley" from TankWar's original model to its 4^{th} evolution step (Figure 5.7). These outliers result from different ordering of variables in ADDs. The inclusion of new variables for the mentioned cases led to a variable ordering that caused a decrease in the number of internal nodes of the resulting ADDs. Thus, the space needed by such data structures was reduced, and so was the time needed to perform ADD operations (which are linear in the number of internal nodes).

Moreover, the approach does not constrain the relation between the feature model's structure and the UML behavioral models implementing the SPL. For instance, the sequence diagram depicted in Figure 2.4b represents optional behavioral fragments that do not follow the structure of the feature model presented by Figure 2.3. The *Oxygenation* feature and the *Persistence* features (*SQLite* and *Memory*) are defined in different branches of the feature model, but the behavioral fragments related to them are nested. In general, the guard condition of an optional behavioral fragment is a propositional formula defined over features and can be defined arbitrarily, with no regard to the structure of the feature model.

Finally, the effect of having (many) cross-tree constraints in a feature model may affect the evaluation method in a twofold manner. First, by adding cross-tree constraints, the structure of the ADD representing the feature model's rules and the reliabilities values of each node is changed. However, it is not possible to foresee if the number of internal nodes will increase, decrease or stay the same, since this number also depends on the variable ordering. In the implementation, such ordering is defined by an internal heuristic defined by the CUDD library, on which the tool relies (namely, *symmetric sifting*). The second effect regards to the growth of the configuration space. In the experiments, the growth in the configuration space at each evolution step will be less than it is now, which will probably have a positive effect in the scalability of the strategies relying on a product-based step. However, since cross-tree constraints would have a random effect on the assessment, it is decided to not add them, so as to have more control over the dependent variables.

5.4 Threats to validity

A threat to internal validity is the creation of UML behavioral models of the product lines by graduate students. To mitigate this threat, the students received an initial training on modeling variable behavior of product lines. To validate the accuracy of the produced models, these were inspected by the research group in which the author is comprised.

A possible threat to construct validity would be an inadequate definition of metrics for the experiment. To address this, it was tried to rule out implementation issues such as the influence of parallelism and reporting of results. Thus, the measure comprises the total elapsed time between the parsing of behavioral models and the instant the reliabilities were ready to be reported, with all analysis steps taking place sequentially. In terms of memory usage, the peak memory usage during execution was measured in order to try to reduce the influence of garbage collection.

Finally, a threat to external validity arises from the selection of subject systems. To mitigate this threat, it were selected systems commonly used by the community as benchmarks to evaluate work on model checking of product lines. To mitigate the risk of the approach not being generalizable, we applied it to further product lines (InterCloud and TankWar) whose configuration spaces resemble ones of real-world applications.

Chapter 6

Conclusion

A feature-family-based method and its corresponding tool for efficient reliability analysis of software product lines were presented. Albeit the reliability evaluation is usually considered in terms of rate of failures, in the context of this work the reliability property is understood in a probabilistic sense. This way, the probability is given by the reachability measure of of a set of successfull states in a probabilistic model enriched with variability (FDTMC). The approach limits the effort needed to compute the reliability of a product line by initially employing a *feature-based* analysis to divide its behavioral models into smaller units, which can be verified more efficiently. For this purpose, the method arranges probabilistic models in an RDG, which is a directed acyclic graph with variability information. This strategy facilitates reuse of reliability computations for redundant behaviors. The *family-based* step comes next when it is performed the reliability computation for all configurations at once by evaluating reliability expressions in terms of ADDs. These decision diagrams encode presence conditions and the rules from the feature model, so that computation is inherently restricted to valid configurations.

The empirical evaluation was accomplished by conducting an experiment to compare the feature-family-based approach with the following evaluation strategies: feature-product-based, family-based, family-product-based, and product-based. Overall, the results show the product-based had the worst time and space performance among all strategies, as expected. The family- and family-product-based strategies yield more complex probabilistic models than the other strategies, due to variability encoding in their models. The product, family-product and feature-product-based approaches were sensitive to the size of the configuration space of the software product line, given their inherent enumerative characteristic. Overall, the experiments show that the feature-family-based strategy is faster than all other analysis strategies and demanded less memory in most cases, being the only one that could be scaled to a 2^{20}-fold increase in the configuration space. Such results suggest that the feature-family-based strategy outperformed the alternative strategies due to the following: (a) the feature-based step explores a lower number of simpler models having fewer variables in comparison to family-based models; and (b) as the family-based step leverages ADD to compute reliability values, fewer operations are necessary to compute reliability values in comparison to the enumerative strategies.

The presented evaluation method is proposed to analyze the software reliability considering such property in a probabilistic manner such the software's behavior depends only on the current state. In this context, the probabilistic software behavior is represented by a Discrete-Time Markov Chain endowed with variability in order to allow representing the behavioral variability of software product lines. However, such evaluation method may also be useful to evaluate other software properties, in special those whose can be represented and evaluated considering probabilistic models. The assumptions for extending the evaluation method for other properties are twofold: a) the software behavioral models must be decomposable into smaller behavioral models (ie. fragments) such there is a runtime dependency between them and b) the same property statemente must be used to evaluate all behavioral fragments. In this sense, it is expected the evaluation method may be adapted to evaluate other software properties, as performance and throughput.

6.1 Future Work

Some investigations are planned to a nearby future. Taking into account the scope of the work hereby presented, it is planned to promptly continue its empirical evaluation. The extension of such evaluation seeks to apply the evaluation method in a larger number of subject systems whom are known to be real world applications (or having much resemblances to real world applications). The intent for such extension is to corroborate the results gathered so far but mainly to investigate the adoption feasibility of the evaluation method by the industry.

Still in the scope of this work it is envisioned the investigation of the evaluation's sensitiveness in terms of the characteristics of the software product line. Such investigation may establish a correlation between the elements of the problem and solution spaces and the effort required by the evaluation. Regarding the problem space, the investigation of each kind of features (mandatory, optional, alternative and exclusive) as well the cross-tree constraints may be useful to define which state-of-the-art evaluation strategy should be employed in order to shorten the evalution effort. Taking the solution space into account, the correlation between the UML behavioral elements and the time and space required for the evaluation may also be bring insights about the their impact over the time and space required by the evaluation. In special, it is known the ADD's size is defined by the number of its internal and terminal nodes and also the number of terminal nodes is directly related to the precision considered for representing the probabilities values. However an investigation addressing the tradeoff between precision and performance (regarding the ADD's size) is still lacking. The impact of other solution space's elements – like number of loop fragments, decision nodes, among others – over the evaluation method's performance must also unveils new opportunities for improving the analysis of software product lines.

Finally, the investigation about adapting the evaluation method to other software's properties may be fruitful. Since the work hereby presented consider the software's reliability in a probabilistic sense by variable Discrete-Time Markov Chains, the immediate investigation to be performed is to evaluate the reliability of software product lines considering the failure's rate of each software component. In this case the variable and probabilistic behavior must be represented by Continuous-Time Markov Chains and the reliability property has to be specified by a CTL statement. Thus, such investigation may bring insights and answers to the characteristics that must be present at the behavioral models that allows performing a feature-family-based analysis. Some assumptions for such characteristics – like the models composability and the presence of runtime dependencies – were addressed throughout the text but an empirical evaluation must refute or corrobate them.

6.2 Related works

This section discusses related work to the hereby presented method, and it is highlighted the significant differences. For this purpose the classification of [9] is considered. The approach differs from prior work [14, 11, 12] in that (a) it captures the runtime feature dependencies from the UML behavioral models, (b) which are enriched with variability information extracted from the feature model, and (c) it leverages ADDs to compute the reliability of all products of a product line with fewer operations than an enumeration would require.

6.2.1 Comparison to a Feature-Product-based Strategy

The evaluation method proposed by [11] is the closest to the work and, to the best of our knowledge, it represents the state-of-the-art for reliability evaluation of software product lines. The whole behavior of a product line is modeled by a set of small sequence diagrams arranged in a tree, where each node has an associated expression resulting from the analysis performed by a parametric model checker. To compute the reliability of a product, the tree is traversed in a bottom-up fashion, when each node's expression is solved considering the configuration under analysis. The resulting value for the root node denotes the product's reliability. This method reduces time and effort required for evaluation by employing parametric in place of classic model checking, but it faces scalability issues as it is inherently enumerative (i.e., the decomposition tree is traversed for each product). The analysis strategy followed by the method is *Feature-Product-based*, as it decomposes the behavioral models into smaller units (feature-based step) and later composes the evaluation results of each unit to obtain the reliability of a product (product-based step).

Despite the resemblances with this method, the approach presents some distinguishing characteristics. While [11] must explore their decomposition tree each time a configuration is evaluated (thus employing a product-based analysis as an evaluation step), the approach employs a family-based evaluation for each RDG node, such that all reliability values it may assume are computed in a single step. Another difference refers to the usage of UML sequence diagram elements for representing behavioral variability. [11] establish a direct relation from the feature model's semantics of optional and alternative features and the semantics of optional (`OPT`) and alternative (`ALT`) combined fragments, respectively. Although such relation is straightforward, it constrains the approach's expressiveness, as only single features can be associated to a combined fragment (i.e., the combined fragment's guard condition assumes only atomic propositions). In contrast, the approach represents behavioral variability uniformly by the optional combined fragment, with an arbitrary presence condition as a guard statement. This construct is simpler, because it does not leverage alternative fragments, but more expressive, as guards can be defined by propositional statements.

Another major difference concerns the underlying data structure for representing the dependencies between behavioral fragments. [11] use a decomposition tree while the approach uses a directed acyclic graph that allows to represent a group of replicated behavioral fragments by a single node. This avoids the effort of performing redundant modeling and evaluation of the replicated model, which is not possible to accomplish in a tree structure.

A precise comparison of the tool implementing the method proposed by Ghezzi and Sharifloo [11] and ReAna was not possible, since the former is not publicly available. Nonetheless, the feature-product-based variant of ReAna created for the experiment closely resembles Ghezzi and Sharifloo's approach, the only exception being the parametric model checker of choice. Empirical results (Section 5.3) show with statistical significance that the feature-family-based approach performs faster and demands less memory than ReAna's feature-product-based variant. For the evaluation time, the feature-family strategy outperformed the feature-product-based strategy from 2 times (for the original seed of EMail system) up to 4 orders of magnitude (for the 3^{rd} evolution of Intercloud product line). Regarding space, the feature-family-based strategy required from 2.6% (original seed of Email system) up to 97% (3^{rd} evolution of InterCloud) less memory. Moreover, the feature-product-based strategy was not able to analyze the subject system with the largest configuration space (Tankwar), whereas the feature-family-based strategy succeeded up to Tankwar's 9^{th} evolution.

Ghezzi and Sharifloo's work [11] presents a theoretical analysis of time complexity, in which the authors devise a formula for computing the time needed to verify a number of properties for a product line with their approach. Their model transformation time is not comparable to this work, mainly because Ghezzi and Sharifloo's[11] do not handle activity diagrams in their work, and this work does not handle reward models in ours. Also, both approaches use external tools with similar capabilities to perform parametric model checking. In fact, the authors argue their tool [48] is actually faster than PARAM, which is used by ReAna. Nonetheless, both model checkers could be used interchangeably, so the parametric model checking time is omitted.

Because of that, it is assumed the output expressions from the model checking phase to be correspondingly equal in both approaches. This way, the difference between the strategies is isolated in the way they solve each expression. While Ghezzi and Sharifloo's [11] perform a number k of floating-point operations for each configuration, the approach performs the same number k of ADD operations, but only once. Since the number of configurations is $O(2^F)$, the feature-product-based approach performs $O(k \cdot 2^F)$ computing steps. As no lowest number of steps is possible if one is to compute the reliability of all possible configurations, the number of computations in the best case is also $O(k \cdot 2^F)$. In contrast, an operation over ADDs in the approach comprises $O(2^{2 \cdot F})$ steps in the worst case, but is $O(F^2)$ in the best case (see Section 5.2). Thus, the feature-family-based approach performs between $O(k \cdot F^2)$ and $O(k \cdot 2^{2 \cdot F})$ computing steps.

Hence, in the worst case, the upper bound for the method's asymptotic complexity is worse than that of [11]'s, but its best-case complexity is better, which is consistent with the empirical findings from the previous chapter.

6.2.2 Other Related Work

[12] present and compare three family-based strategies to analyze probabilistic properties of product lines. Two of them leverage PARAM as model checker; the third one relies on FDTMCs representing the behavior of a whole product line by encoding its variability, resulting in an ADD expressing the reliability values of all configurations. The feature-family-based strategy benefits even more from further breaking down probabilistic models. Indeed, the methods by Rodrigues et. al.[12] show a time-space tradeoff, but all of them presented scalability issues even for small product lines (around 12 features), whereas the

approach is able to analyze a product line with 144 features and about 10^{18} products within reasonable time.

Further research has addressed efficient verification of other non-functional properties of product lines by exploiting family-based analysis strategies [54, 17, 16, 55, 56, 57, 14, 58]. [54] propose an approach for performance evaluation by simulating the behavior of all variants at runtime from the variability encoded in compile-time. Such simulator is created from the log of method calls traced by features. [17] create a model representing the whole performance variability of a product line from UML activity diagrams annotated with performance-related annotations. [16, 55] present an approach for modeling dynamic product lines and performing quantitative analysis of systems endowed of non-deterministic choices. Given the non-deterministic characteristic of the systems evaluated by this approach, the authors consider Markov Decision Processes as the suitable model for representing the model behavior. Similarly, [56] introduce a mathematical model named Markov Decision Process Family for representing the behavior of a product line as a whole, as well as a model checking algorithm to verify properties expressed in probabilistic computation tree logic. [14] establish the foundations of *Featured Transition Systems* (FTS) to create a model endowed with features expressions to represent the states variation of the whole software product line. The authors also present a family-based model checker [57] that is able to analyze *Linear Temporal Logic* (LTL) properties of the whole software product line by employing semi-symbolic algorithms to verify FTSs. All these pieces of work exploit symbolic computation on a model representing the whole variability of a product line as a better alternative to product-based strategies. The study supports this conclusion, especially if a suitable variational data structure (e.g., ADD) is used for such analysis. However, the results indicate that feature-family-based analysis further improves performance.

[58] also present an efficient family-based technique to verify LTL properties of a software family. The authors leverage abstract interpretation to reduce the configuration space of an FTS, so that it can be verified by off-the-shelf model checkers (i.e., aimed and optimized to analyze single systems). The method employs a divide-and-conquer strategy to reduce model size, without changing the configuration space. Moreover, the analysis method also employs off-the-shelf model checkers, but to analyze probabilistic properties of software product lines. Therefore, it is worth investigating the extent to which the technique proposed by [58] can be applied to the verification of PCTL properties. If that is the case, we conjecture that both strategies could be combined to further reduce verification effort.

Referências

[1] Pohl, Klaus, Günter Böckle e Frank van der Linden: *Software product line engineering: foundations, principles, and techniques*. Springer, New York, NY, 2010. 1

[2] Clements, Paul e Linda Northrop: *Software product lines: practices and patterns*. The SEI series in software engineering. Addison-Wesley, Boston, 2002, ISBN 978-0-201-70332-0. 1, 15

[3] Czarnecki, Krzysztof e Ulrich Eisenecker: *Generative programming: methods, tools, and applications*. Addison Wesley, Boston, 2000, ISBN 978-0-201-30977-5. 1, 15

[4] Weiss, David M.: *The product line hall of fame*. Em *Proceedings of the 12th International Software Product Line Conference (SPLC)*, página 395, Washington, DC, USA, 2008. IEEE Computer Society, ISBN 978-0-7695-3303-2. 1

[5] Linden, Frank van der, Klaus Schmid e Eelco Rommes: *Software product lines in action: the best industrial practice in product line engineering*. Springer, Berlin ; New York, 2007. 1, 15

[6] Apel, Sven, Don Batory, Christian Kästner e Gunter Saake (editores): *Feature-oriented software product lines: concepts and implementation*. Springer, Berlin, 2013, ISBN 978-3-642-37521-7 978-3-642-37520-0. 1, 2, 15

[7] Heradio, Ruben, Hector Perez-Morago, David Fernandez-Amoros, Francisco Javier Cabrerizo e Enrique Herrera-Viedma: *A bibliometric analysis of 20 years of research on software product lines*. Information and Software Technology, 72:1–15, abril 2016. 1

[8] Machado, Ivan do Carmo, John D. McGregor, Yguaratã Cerqueira Cavalcanti e Eduardo Santana de Almeida: *On strategies for testing software product lines: A systematic literature review*. Information and Software Technology, 56(10):1183–1199, outubro 2014. 1

[9] Thüm, Thomas, Sven Apel, Christian Kästner, Ina Schaefer e Gunter Saake: *A Classification and Survey of Analysis Strategies for Software Product Lines*. ACM Comput. Surv., 47(1):6:1–6:45, junho 2014. 1, 2, 4, 5, 15, 16, 52, 70, 83

[10] Baier, Christel e Joost Pieter Katoen: *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008, ISBN 026202649X, 9780262026499. 1, 2, 9, 10, 47, 69

[11] Ghezzi, Carlo e Amir Molzam Sharifloo: *Model-based verification of quantitative non-functional properties for software product lines*. Information and Software Technology, 55(3):508–524, março 2013. 2, 3, 4, 61, 83, 84, 85

[12] Rodrigues, Genaina Nunes, Vander Alves, Vinicius Nunes, Andre Lanna, Maxime Cordy, Pierre Yves Schobbens, Amir Molzam Sharifloo e Axel Legay: *Modeling and Verification for Probabilistic Properties in Software Product Lines*. Em *2015 IEEE 16th International Symposium on High Assurance Systems Engineering (HASE)*, páginas 173–180, janeiro 2015. 2, 5, 17, 53, 71, 83, 85

[13] Classen, Andreas, Maxime Cordy, Patrick Heymans, Axel Legay e Pierre Yves Schobbens: *Formal semantics, modular specification, and symbolic verification of product-line behaviour*. Science of Computer Programming, 80, Part B:416–439, fevereiro 2014, ISSN 0167-6423. 2, 71

[14] Classen, A., M. Cordy, P. Y. Schobbens, P. Heymans, A. Legay e 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. 2, 4, 15, 71, 83, 86

[15] Classen, A., P. Heymans, P. Schobbens e A. Legay: *Symbolic model checking of software product lines*. Em *2011 33rd International Conference on Software Engineering (ICSE)*, páginas 321–330, 2011. 2

[16] Dubslaff, Clemens, Christel Baier e Sascha Kluppelholz: *Probabilistic Model Checking for Feature-Oriented Systems*. Em Chiba, Shigeru, Eric Tanter, Erik Ernst e Robert Hirschfeld (editores): *Transactions on Aspect-Oriented Software Development XII*, número 8989 em *Lecture Notes in Computer Science*, páginas 180–220. Springer Berlin Heidelberg, 2015, ISBN 978-3-662-46733-6 978-3-662-46734-3. DOI: 10.1007/978-3-662-46734-3_5. 2, 86

[17] Kowal, Matthias, Max Tschaikowski, Mirco Tribastone e Ina Schaefer: *Scaling Size and Parameter Spaces in Variability-Aware Software Performance Models (T)*. Em *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, páginas 407–417, novembro 2015. 2, 86

[18] Nunes, V., P. Fernandes, V. Alves e G. Rodrigues: *Variability Management of Reliability Models in Software Product Lines: An Expressiveness and Scalability Analysis*. Em *2012 Sixth Brazilian Symposium on Software Components Architectures and Reuse (SBCARS)*, páginas 51–60, setembro 2012. 2

[19] Clarke, E. M., Orna Grumberg e Doron A. Peled: *Model checking*. MIT Press, Cambridge, Mass, 1999, ISBN 978-0-262-03270-4. 2

[20] Classen, Andreas, Patrick Heymans, Pierre Yves Schobbens, Axel Legay e Jean François Raskin: *Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines*. Em *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, páginas 335–344, New York, NY, USA, 2010. ACM, ISBN 978-1-60558-719-6. 2

[21] Bodden, Eric, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba e Mira Mezini: *SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years*. Em *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, páginas 355–364, New York, NY, USA, 2013. ACM, ISBN 978-1-4503-2014-6. 2

[22] Avizienis, A., J. C. Laprie, B. Randell e C. Landwehr: *Basic concepts and taxonomy of dependable and secure computing*. IEEE Transactions on Dependable and Secure Computing, 1(1):11–33, janeiro 2004, ISSN 1545-5971. 2

[23] Grunske, Lars: *Specification patterns for probabilistic quality properties*. Em *ICSE '08*, páginas 31–40, New York, NY, USA, 2008. ACM. 2, 8, 22, 47

[24] Shin, K.G. e P. Ramanathan: *Real-time computing: a new discipline of computer science and engineering*. Proceedings of the IEEE, 82(1):6–24, janeiro 1994, ISSN 0018-9219. 3

[25] Rodrigues, Genaína Nunes, Vander Alves, Renato Silveira e Luiz A. Laranjeira: *Dependability analysis in the Ambient Assisted Living Domain: An exploratory case study*. Journal of Systems and Software, 85(1):112–131, janeiro 2012, ISSN 0164-1212. 3

[26] Kästner, Christian, Sven Apel e Martin Kuhlemann: *Granularity in software product lines*. Em *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, páginas 311–320, 2008. `http://doi.acm.org/10.1145/1368088.1368131`. 4, 15

[27] Bahar, R.I., E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo e F. Somenzi: *Algebric decision diagrams and their applications*. Formal Methods in System Design, 10(2):171–206, 1997, ISSN 1572-8102. 4

[28] Kramer, J., J. Magee, M. Sloman e A. Lister: *CONIC: an integrated approach to distributed computer control systems*. Computers and Digital Techniques, IEE Proceedings E, 130(1):1–, janeiro 1983. 5, 71

[29] Plath, Malte e Mark Ryan: *Feature integration using a feature construct*. Science of Computer Programming, 41(1):53–84, setembro 2001. 5, 71

[30] University of Magdeburg, Otto von Guericke: *SPL2go*. Available at `http://spl2go.cs.ovgu.de/`, 2011. Accessed: 2016-01-27. 5, 71

[31] Ferreira Leite, A., V. Alves, G. Nunes Rodrigues, C. Tadonki, C. Eisenbeis e A.C. Magalhaes Alves de Melo: *Automating Resource Selection and Configuration in Inter-clouds through a Software Product Line Method*. Em *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, páginas 726–733, junho 2015. 5, 71

[32] Kwiatkowska, M., G. Norman e D. Parker: *PRISM 4.0: Verification of probabilistic real-time systems*. Em *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 de *Lecture Notes in Computer Science*, páginas 585–591. Springer, 2011. 9, 10

[33] Daws, Conrado: *Symbolic and parametric model checking of discrete-time Markov chains*. Em *Proceedings of the First International Conference on Theoretical Aspects of Computing (ICTAC)*, volume 3407 de *Lecture Notes in Computer Science*, páginas 280–294. Springer, sep 2005, ISBN 978-3-540-25304-4. 9, 10

[34] Grunske, Lars: *Specification patterns for probabilistic quality properties*. Em *Proceedings of the International Conference on Software Engineering (ICSE)*, páginas 31–40. ACM, 2008. 9

[35] Rodrigues, Genaína Nunes, Vander Alves, Vinicius Nunes, André Lanna, Maxime Cordy, Pierre-Yves Schobbens, Amir Molzam Sharifloo e Axel Legay: *Modeling and verification for probabilistic properties in software product lines*. Em *Proceedings of the 16th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, páginas 173–180. IEEE Computer Society, 2015. 10

[36] Ghezzi, Carlo e Amir Molzam Sharifloo: *Model-based verification of quantitative non-functional properties for software product lines*. Information and Software Technology, 55(3):508–524, março 2013, ISSN 09505849. 10

[37] Chrszon, Philipp, Clemens Dubslaff, Sascha Klüppelholz e Christel Baier: *Family-based modeling and analysis for probabilistic systems - featuring ProFeat*. Em *Proceedings of the 19th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 9633 de *Lecture Notes in Computer Science*, páginas 287–304. Springer, 2016. 10

[38] Hahn, Ernst Moritz, Holger Hermanns, Björn Wachter e Lijun Zhang: *Probabilistic reachability for parametric markov models*. STTT, páginas 1–17, 2010. 10, 11, 12, 13, 15, 52, 68

[39] Bahar, R. Iris, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo e Fabio Somenzi: *Algebraic decision diagrams and their applications*. Formal Methods in System Design, 10(2/3):171–206, 1997. 13, 14

[40] Kang, K., S. Cohen, J. Hess, W. Novak e S. Peterson: *Feature-oriented domain analysis (FODA) feasibility study*. Relatório Técnico CMU/SEI-90-TR-21, Carneggie Mellon University, 1990. 15

[41] Daws, Conrado: *Symbolic and Parametric Model Checking of Discrete-time Markov Chains*. Em Liu, Zhiming e Keijiro Araki (editores): *Proceedings of the First International Conference on Theoretical Aspects of Computing*, volume 3407 de *Lecture Notes in Computer Science*, páginas 280–294, Berlin, Heidelberg, sep 2005. Springer Berlin Heidelberg, ISBN 978-3-540-25304-4. 15, 45, 48

[42] Kwiatkowska, M., G. Norman, D. Parker e H. Qu: *Compositional probabilistic verification through multi-objective model checking*. Information and Computation, 232:38–65, 2013. 16

[43] Hao, Yang e Robert Foster: *Wireless body sensor networks for health-monitoring applications*. Physiological Measurement, 29(11):27–56, 2008. 17

[44] Czarnecki, Krzysztof e Krzysztof Pietroszek: *Verifying feature-based model templates against well-formedness OCL constraints*. Em *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, página 211. ACM Press, 2006, ISBN 978-1-59593-237-2. 18, 54

[45] Pessoa, Leonardo, Paula Fernandes, Thiago Castro, Vander Alves, Genaína N. Rodrigues e 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. 20

[46] Object Management Group: *The UML profile for MARTE: Modeling and analysis of real-time and embedded systems*. http://www.omg.org/omgmarte/, 2011. Version 1.1. 21, 53

[47] Hahn, Ernst Moritz, Holger Hermanns, Björn Wachter e Lijun Zhang: *Param: A model checker for parametric markov models*. Em *CAV*, páginas 660–664, 2010. 59, 66

[48] Filieri, Antonio e Carlo Ghezzi: *Further steps towards efficient runtime verification: Handling probabilistic cost models*. Em *Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*, FormSERA '12, páginas 2–8, Piscataway, NJ, USA, 2012. IEEE Press, ISBN 978-1-4673-1906-5. 59, 66, 85

[49] Walkingshaw, Eric, Christian Kästner, Martin Erwig, Sven Apel e Eric Bodden: *Variational Data Structures: Exploring Tradeoffs in Computing with Variability*. Em *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, páginas 213–226, New York, NY, USA, 2014. ACM. 61

[50] Iris, R., Bahar Erica, A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo e Fabio Somenzi: *Algebraic Decision Diagrams and Their Applications*. Em *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design (ICCAD '93)*, páginas 188–191, Santa Clara, California, USA, 1993. IEEE Computer Society Press. 61, 69

[51] Thüm, Thomas, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake e Thomas Leich: *Featureide: An extensible framework for feature-oriented software development*. Sci. Comput. Program., 79:70–85, janeiro 2014, ISSN 0167-6423. 66

[52] Cormen, Thomas H., Clifford Stein, Ronald L. Rivest e Charles E. Leiserson: *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edição, 2001, ISBN 0070131511. 68

[53] Liang, Jia Hui, Vijay Ganesh, Krzysztof Czarnecki e Venkatesh Raman: *SAT-based Analysis of Large Real-world Feature Models is Easy*. Em *Proceedings of the 19th International Conference on Software Product Line*, SPLC '15, páginas 91–100, New York, NY, USA, 2015. ACM, ISBN 978-1-4503-3613-0. 69

[54] Siegmund, Norbert, Alexander von Rhein e Sven Apel: *Family-based Performance Measurement*. Em *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, páginas 95–104, New York, NY, USA, 2013. ACM, ISBN 978-1-4503-2373-4. 86

[55] Dubslaff, Clemens, Sascha Klüppelholz e Christel Baier: *Probabilistic Model Checking for Energy Analysis in Software Product Lines*. Em *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, páginas 169–180, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2772-5. 86

[56] Varshosaz, M. e R. Khosravi: *Model Checking of Software Product Lines in Presence of Nondeterminism and Probabilities*. Em *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 1, páginas 63–70, dezembro 2014. 86

[57] Classen, Andreas, Maxime Cordy, Patrick Heymans, Axel Legay e Pierre Yves Schobbens: *Model checking software product lines with SNIP*. International Journal on Software Tools for Technology Transfer, 14(5):589–612, Oct 2012, ISSN 1433-2787. 86

[58] Dimovski, Aleksandar S., Ahmad Salim Al-Sibahi, Claus Brabrand e Andrzej Wąsowski: *Family-Based Model Checking Without a Family-Based Model Checker*, páginas 282–299. Springer International Publishing, Cham, 2015, ISBN 978-3-319-23404-5. 86

[59] Mendonca, Marcilio, Moises Branco e Donald Cowan: *S.p.l.o.t.: Software product lines online tools*. Em *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, páginas 761–762, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-768-4. 95

Appendix A

Experiment Data

The following tables present the mean values for analysis time and memory consumption obtained in our experiment. Values typeset in boldface are the best values (i.e., the lowest) gathered from the experiments. Cells containing dashes represent unavailable data, meaning that the corresponding analysis violated the time limit of 60 minutes.

Table 1: Time in milliseconds (fastest strategy in boldface).

		SPL's evolutions steps										
		0 10	1 11	2 12	3 13	4 14	5 15	6 16	7 17	8 18	9 19	 20
Email	*Configuration space's order*	10^1	10^1	10^2	10^2	10^2	10^3	10^3	10^3	10^4	10^4	
	Feature-family	**183.04**	**223.78**	**233.69**	**259.65**	**267.32**	**285.79**	**341.65**	**348.46**	**366.73**	**433.30**	
	Feature-product	370.63	517.67	742.91	1108.95	1659.31	2358.51	3829.95	6919.98	12803.15	25110.63	
	Family	319.72	1167.27	13944.18	154067.34	–	–	–	–	–	–	
	Family-product	293.26	558.77	1095.75	2850.86	9451.57	34704.42	137866.42	562117.02	1607837.42	–	
	Product	2424.77	7387.35	14349.32	29137.45	57575.0	114084.61	275598.17	–	–	–	
	Configuration space's order	10^4	10^4	10^5	10^5	10^5	10^6	10^6	10^6	10^7	10^7	10^7
	Feature-family	**970.69**	**1613.76**	**2833.40**	**5425.14**	**10838.39**	**21719.17**	**44171.89**	**90015.26**	**187645.77**	**667138.0**	–
	Feature-product	50748.90	103510.61	215932.90	456329.22	945445.46	1966865.48	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
Minepump	*Configuration space's order*	10^2	10^2	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	
	Feature-family	**261.18**	**287.24**	**298.10**	**330.88**	**358.81**	**408.45**	**485.06**	**621.01**	**877.52**	**1375.22**	
	Feature-product	895.80	1226.97	1844.15	2624.96	4204.27	8952.61	14037.50	25989.10	51495.22	104090.89	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^5	10^5	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8
	Feature-family	**2390.78**	**4445.44**	**8790.54**	**17995.17**	**36593.45**	**76513.51**	**168694.38**	**354887.72**	–	–	–
	Feature-product	211806.42	439411.29	905878.46	1876640.52	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
BSN	*Configuration space's order*	10^2	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	10^5	
	Feature-family	**237.14**	**253.65**	**273.01**	**305.48**	**321.69**	**377.40**	**389.41**	**462.66**	**651.84**	**1032.05**	
	Feature-product	991.30	1487.19	2404.18	4312.01	7875.91	14788.91	28881.71	57887.92	117630.81	241553.61	
	Family	1604.07	–	–	–	–	–	–	–	–	–	
	Family-product	3288.70	11543.38	46273.48	187134.89	672512.22	2109118.92	–	–	–	–	
	Product	6696.06	20259.05	43489.98	97280.21	241249.72	519495.92	1217404.53	–	–	–	
	Configuration space's order	10^5	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8	10^8
	Feature-family	**1713.19**	**3443.81**	**7332.75**	**15090.83**	**31208.01**	**83984.25**	**197660.21**	**528948.31**	–	–	–
	Feature-product	495594.45	1022294.56	2145986.30	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–

continued in the next page

continued from last page

		SPL's evolutions steps										
		0 10	1 11	2 12	3 13	4 14	5 15	6 16	7 17	8 18	9 19	20
Lift	*Configuration space's order*	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	10^5	10^5	
	Feature-family	**140.32**	**169.51**	**188.56**	**199.95**	**223.20**	**266.20**	**339.85**	**472.01**	**601.46**	**1021.76**	
	Feature-product	1160.78	1786.75	1289.76	4281.11	7739.10	14769.15	29418.50	60785.39	127344.46	266609.58	
	Family	358.06	625.16	2606.86	18223.06	–	–	–	–	–	–	
	Family-product	1413.96	3462.52	10777.60	42156.47	167837.42	453830.76	1870142.24	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8	10^8	10^9
	Feature-family	**3114.06**	**5728.56**	**10895.96**	**21081.40**	**48194.05**	**100755.69**	**215756.37**	–	–	–	–
	Feature-product	555525.51	1136506.33	2457317.34	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
InterCloud	*Configuration space's order*	10^5	10^5	10^5	10^5	10^6	10^6	10^6	10^7	10^7	10^7	
	Feature-family	**671.54**	**717.7**	**794.95**	**880.11**	**922.98**	**994.65**	**1126.91**	**1315.89**	**1742**	**2544.49**	
	Feature-product	407702.34	861181.31	1752682.98	3394277.78	–	–	–	–	–	–	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^8	10^8	10^8	10^8	10^9	10^9	10^9	10^{10}	10^{10}	10^{10}	10^{11}
	Feature-family	**4074.43**	**4280.4**	**4568.7**	**5344.4**	**3936.76**	**6719.68**	**12829.35**	**25588.69**	**67156.86**	–	–
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
TankWar	*Configuration space's order*	10^{18}	10^{18}	10^{19}	10^{19}	10^{19}	10^{20}	10^{20}	10^{20}	10^{21}	10^{21}	
	Feature-family	**6643.88**	**3588.49**	**2734.86**	**2966.2**	**2902.18**	**3079.4**	**4221.14**	**8012.**	**17096.88**	**160259.19**	
	Feature-product	–	–	–	–	–	–	–	–	–	–	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^{21}	10^{21}	10^{22}	10^{22}	10^{22}	10^{23}	10^{23}	10^{23}	10^{24}	10^{24}	10^{24}
	Feature-family	–	–	–	–	–	–	–	–	–	–	–
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–

Table 2: Space in megabytes (smallest footprint in boldface).

		SPL's evolutions steps										
		0 10	1 11	2 12	3 13	4 14	5 15	6 16	7 17	8 18	9 19	20
Email	*Configuration space's order*	10^1	10^1	10^2	10^2	10^2	10^3	10^3	10^3	10^4	10^4	
	Feature-family	**113.70**	**113.84**	**113.93**	**114.30**	**114.45**	**114.33**	**114.52**	**114.91**	**115.86**	**117.64**	
	Feature-product	117.22	144.30	186.59	269.67	475.61	738.99	1136.73	2359.24	2839.02	2842.46	
	Family	116.97	125.48	136.57	196.99	–	–	–	–	–	–	
	Family-product	120.25	157.90	235.41	510.41	827.79	722.88	1037.62	1501.80	3231.31	–	
	Product	122.65	231.84	272.04	277.98	310.59	309.06	327.65	–	–	–	
	Configuration space's order	10^4	10^4	10^5	10^5	10^5	10^6	10^6	10^6	10^7	10^7	10^7
	Feature-family	**130.65**	**146.25**	**174.93**	**287.00**	**489.00**	**839.80**	**1523.88**	**3041.86**	**5807.80**	**7223.00**	–
	Feature-product	2849.01	2878.10	2927.46	3158.43	3367.68	4181.64	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
MinePump	*Configuration space's order*	10^2	10^2	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	
	Feature-family	**113.51**	**114.05**	**114.41**	**114.34**	**114.8**	**115.61**	**116.47**	**118.48**	**129.12**	**133.96**	
	Feature-product	210.97	333.42	504.98	743.93	1319.2	2400.89	2841.77	2844.10	2851.49	2879.39	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^5	10^5	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8
	Feature-family	**162.48**	**265.31**	**390.03**	**705.39**	**1165.72**	**2224.17**	**4011.27**	**6921.67**	–	–	–
	Feature-product	2914.44	2971.55	3378.84	3789.31	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
BSN	*Configuration space's order*	10^2	10^2	10^3	10^3	10^3	10^3	10^4	10^4	10^4	10^5	
	Feature-family	**114.05**	**114.30**	**114.52**	**114.56**	**114.83**	**115.37**	**115.32**	**116.97**	**120.09**	**134.23**	
	Feature-product	339.91	490.76	737.50	1716.41	2379.06	2837.60	2843.36	2850.78	2874.49	2923.93	
	Family	156.54	–	–	–	–	–	–	–	–	–	
	Family-product	493.99	841.31	1171.71	1153.13	2189.89	3263.80	–	–	–	–	
	Product	320.43	335.18	339.40	352.72	327.95	440.60	446.75	–	–	–	
	Configuration space's order	10^5	10^5	10^6	10^6	10^6	10^6	10^7	10^7	10^7	10^8	10^8
	Feature-family	**148.34**	**186.03**	**348.58**	**588.99**	**1043.94**	**2225.13**	**4640.35**	**7130.79**	–	–	–
	Feature-product	3005.12	3234.19	3821.39	–	–	–	–	–	–	–	–
	Family	156.54	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–

continued in the next page

continued from last page

		SPL's evolutions steps										
		0 10	1 11	2 12	3 13	4 14	5 15	6 16	7 17	8 18	9 19	20
Lift	*Configuration space's order*	10^{2}	10^{3}	10^{3}	10^{3}	10^{3}	10^{4}	10^{4}	10^{4}	10^{5}	10^{5}	
	Feature-family	**113.77**	**113.85**	**114.37**	**114.27**	**114.56**	**115.23**	**116.87**	**119.88**	**120.72**	**134.41**	
	Feature-product	292.23	507.43	757.52	1539.24	2399.04	2838.97	2840.51	2859.23	2907.41	2993.05	
	Family	116.54	122.63	136.83	177.02	–	–	–	–	–	–	
	Family-product	272.85	506.39	1277.44	1296.95	1551.49	2440.83	2669.75	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^{5}	10^{6}	10^{6}	10^{6}	10^{6}	10^{7}	10^{7}	10^{7}	10^{8}	10^{8}	10^{7}
	Feature-family	**203.42**	**319.45**	**539.66**	**826.57**	**1791.86**	**3230.47**	**6324.48**	–	–	–	–
	Feature-product	3199.10	3489.45	4644.73	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
InterCloud	*Configuration space's order*	10^{5}	10^{5}	10^{5}	10^{5}	10^{6}	10^{6}	10^{6}	10^{7}	10^{7}	10^{7}	
	Feature-family	**119.44**	**119.87**	**127.68**	**127.79**	**136.03**	**132.63**	**136.3**	**143.96**	**152.61**	**175.78**	
	Feature-product	3071.58	3158.59	3602.16	4884.81	–	–	–	–	–	–	
	Family	–	–	–	–	–	–	–	–	–	–	
	Family-product	–	–	–	–	–	–	–	–	–	–	
	Product	–	–	–	–	–	–	–	–	–	–	
	Configuration space's order	10^{8}	10^{8}	10^{8}	10^{8}	10^{9}	10^{9}	10^{9}	10^{10}	10^{10}	10^{10}	10^{11}
	Feature-family	**224.06**	**223.7**	**251.48**	**275.81**	**237.67**	**378.87**	**635.76**	**1102.48**	**2628.21**	–	–
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
TankWar	*Configuration space's order*	10^{18}	10^{18}	10^{19}	10^{19}	10^{19}	10^{20}	10^{20}	10^{20}	10^{21}	10^{21}	
	Feature-family	**286.99**	**258.64**	**246.91**	**256.09**	**253.42**	**271.44**	**295.76**	**407.85**	**622.82**	**4104.61**	
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–
	Configuration space's order	10^{21}	10^{21}	10^{22}	10^{22}	10^{22}	10^{23}	10^{23}	10^{23}	10^{24}	10^{24}	10^{24}
	Feature-family	–	–	–	–	–	–	–	–	–	–	–
	Feature-product	–	–	–	–	–	–	–	–	–	–	–
	Family	–	–	–	–	–	–	–	–	–	–	–
	Family-product	–	–	–	–	–	–	–	–	–	–	–
	Product	–	–	–	–	–	–	–	–	–	–	–

Appendix B

SPL Generator Tool

To increase the number of subject systems and inspect how each evaluation strategy behaves with the growth of the configuration space, we implemented a product-line generator tool called SPL–Generator[1], which is able to create a software product line from scratch or modify an existing one by incrementally adding features and behavior to its models. For the feature model generation (i.e., to create a new feature model or change an existing one), the tool relies on the SPLAR tool [59]. The desired characteristics of the resulting feature model are obtained by defining accordingly the set of parameters provided by SPLAR. Examples of such parameters are the number of features to be created, the amount in percentage for each kind of feature (mandatory, optional, OR-inclusive and OR-exclusive), and the number of cross-tree constraints. As our SPL-Generator tool intends to create product lines that resemble real-world product lines, it produces only consistent feature-models (i.e., the SPLAR's parameter for creating consistent feature-models is always set to `true`).

To create behavioral models, the SPL-Generator tool considers the UML behavioral diagrams and follows the refinement of activity diagrams into sequence diagrams presented in Section 2.5. For creating activity and sequence diagrams, the generator tool is also guided by a set of parameters for each kind of behavioral diagram. For an activity diagram, it is possible to define how many activities it will comprise, the number of decision nodes, and how many sequence diagrams will refine each created activity. For a sequence diagram, it is possible to define its size in terms of numbers of behavioral fragments, the size of each behavioral fragment in terms of the number of messages, the number of lifelines, the number of different reliability values (such that each lifeline will randomly assume only one value) and the range for them. Thus, one possibly generated sequence diagram would have 5 behavioral fragments, each one containing 8 messages between 3 lifelines, whose reliability values are within the range $[0.99, 0.999]$.

[1] https://github.com/SPLMC/spl-generator/

Finally, the SPL-Generator tool also provides a parameter to define how the feature model and the behavioral models will be related. The allocation of a behavioral fragment (implementing a feature's behavior) can be fully *randomized* within the set of created sequence diagrams, or it can be *topological*, which means the relations between the behavioral fragments mimic the relations between the corresponding features. In the latter, we assume a child feature refines its parent, so its behavioral fragment is nested into its parent's behavioral fragment.