POLITECNICO DI MILANO
DEPARTMENT OF ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

REQUIREMENTS VERIFICATION OF VARIABILITY-INTENSIVE SYSTEMS

Doctoral Dissertation of:
Amir Molzam Sharifloo

Supervisor:
Prof. Carlo Ghezzi

Tutor:
Prof. Gian Paolo Cugola

The Chair of the Doctoral Program:
Prof. Carlo Fiorini

2014 - IIV

Acknowledgement

My special thanks goes to my supervisor Prof. Carlo Ghezzi, who provided me with a continuous support through all days of my PhD. I am thankful of Prof. Axel Van Lamsweerde for his insightful comments on the first draft. I have been grateful to work with Paola Spoletini and Antonio Filieri, who generously shared their expertise with me. Moreover, I would like to thank Prof. Patrick Heymans (University of Namur), Emmanuele Letier (University College London), Axel Legay (Inria Rennes), and Andreas Medzger (Universitat Duisburg-Essen), for their hospitality during the periods I visited their research groups.

I was grateful to do my PhD in a big research group DEEPSE, where people go beyond work relations and build good friendships. Thus, I am honored to remember these names: Danilo Ardagna, Elisabetta Di Nitto, Raffaella Mirandola, Alessandra Viale, Luciano Baresi, Mauro Caporuscio, Salvatore Distefano, Matteo Pradella, Marcello Bersani, Michele Ciavotta, Daniel J. Dubois, Andrea Mocci, Alessandro Margara, Liliana Pasquale, Marco Funaro, Joel Greenyer, Giordano Tamburrelli, Luca Cavalaro, Mikhail Afanasov, Nicoló Calcavecchia, Luca Ferrucci, Srdjan Krstic, Santo Lombardo, Claudio Menghi, Alfredo Motta, Diego Perez, Federica Panella, Valerio Panzica La Manna, Mehdi Pourhashem, Mario Sangiorgio, Alessandro Sivieri, Leandro Sales, Matteo Miraz, Guido Salvaneschi, Manuel Mazzara, Pantea Saeedi, Yu Zhou, Vania Neves, Pasqualina Potena, Andrea Ciancone, Giovanni Gibilisco, Marco Miglierina, Alessandro Rizzi.

To my parents for lifelong support,

to my sisters for lifelong company,

to my brother-in-law for our brotherhood,

to my girlfriend for being there,

and

to patience, enthusiasm, and continuation.

Abstract

The objective of this thesis is to devise techniques to verify different properties of Variability-Intensive Systems. By Variability-Intensive System, we refer to a specification from which various systems can be derived. Adaptive systems and software product lines are two important examples of such systems. In the former case, system switches among different configurations at run time, while in the later case, each valid configuration is released as a single product at development time. Incomplete specifications of system behavior, which are incrementally developed during development, are another kind of specifications that evolve over time, and may lead to different releases. We classify variability in two classes: *open* and *closed*. Accordingly, the thesis is divided into two main parts, which are dedicated to these two types of systems. Open variability refers to the cases where the specifications of alternatives are unknown, which is the case of incomplete specifications. This is exactly opposite in the case of closed variability for which the specification is complete, that is the case of software product lines.

Part II presents a novel approach for specification and verification of incomplete software models against temporal logic properties and in particular CTL. An incomplete model may represent a partial design of a system at early development stages, in which some components are not yet specified. The unknown components can be viewed as variation points of such specification, and can be bound with different components depending on later design decisions. Similarly, an adaptive system with multiple variation points may be represented as an incomplete specification, in which

variability is replaced with appropriate alternatives. We show the applicability of this approach in the context of Statecharts and Labeled Transition Systems and provide algorithms and tool support.

Part III focuses on verification of stochastic software product lines (SPLs) for non-functional properties, namely reliability and energy consumption. We propose two modeling approaches to capturing stochastic behavior of SPLs. One approach enriches UML Sequence Diagrams with variability and stochastic information to represent high-level system scenarios. Second modeling, instead, extends Markov models with variability elements. Moreover, we propose three model-checking algorithms for the latter formalism, and discuss their performance and application. Finally we customize our framework to build model-based adaptive systems, which can continuously monitor, check, and satisfy non-functional requirements. In this case, we discuss the application of Dynamic SPLs.

Sommario

L'OBIETTIVO di questa tesi è sviluppare tecniche per verificare diverse proprietà dei Variability-Intensive Systems.

La prima parte si occupa con gli definizioni base che serve per seguire altre due parti. Con Variability-Intensive Systems, ci riferiamo ad una specifica dalla quale i vari sistemi possono essere derivate. Adaptive Systems e Software Product Lines (SPL) sono due importanti esempi di tali sistemi. Nel primo caso, il sistema cambia tra diverse configurazioni in fase di esecuzione. Nel secondo caso ogni configurazione valida è rilasciato e come un singolo prodotto in fase di sviluppo. Specifiche incomplete di comportamento del sistema, che sono progressivamente sviluppate durante lo sviluppo, sono un altro tipo di specifiche che si evolvono nel tempo, e possono portare a versioni differenti. Classifichiamo la variabilità in due classi: *aperta* e *chiusa*. Di conseguenza, la tesi è divisa in due parti principali, che sono dedicati a questi due tipi di sistemi. La variabilità aperta riguarda i casi in cui le specifiche di altre alternative sono sconosciute, che è il caso di specifiche incompleti. Questo è esattamente opposta in caso delle SPL che la variabilità è chiusa e la specifica è completa.

La seconda parte presenta un nuovo approccio per la specifica e la verifica di modelli di software incompleti rispetto le proprietà logiche temporali e particolarmente CTL. Un modello incompleto può rappresentare un disegno parziale di un sistema nelle prime fasi del sviluppo, in cui alcuni componenti non sono ancora stati specificati. I componenti sconosciuti possono essere visti come punti di variazione di tale specifica, e possono essere associati con diversi componenti a seconda delle decisioni di progettazione successivi. Analogamente, un sistema adattivo con più punti di variazione può essere rappresentato come una specifica incompleta, in cui la variabilità è sostituita con alternative appropriate. Mostriamo l'applicabilità di questo approccio nel contesto dagli Statecharts e Labeled Transition Systems e forniamo algoritmi e strumenti di supporto.

La terza parte si concentra sulla verifica delle linee di prodotti software stocastico per requisiti non-funzionali, vale a dire affidabilita e il consumo di energia. Proponiamo due approcci di modellazione per rappresentare il comportamento stocastico di SPL. Il nostro approccio arricchisce diagrammi di sequenza UML con punti di variabilità e informazioni stocastici a rappresentare scenari di sistemi di alto livello. Inoltre la modellazione estende i modelli di Markov con elementi di variabilità. Inoltre, proponiamo tre algoritmi di controllo per quest'ultimo formalismo, e discutiamo le loro prestazioni e applicazioni. Infine abbiamo personalizzato il nostro framework per costruire Adaptive Systems basati su modelli, che possono monitorare, controllare e soddisfare i requisiti non-funzionali. In questo caso, si discute l'applicazione della dinamica alla SPL.

Contents

Publication

The content of this thesis is largely taken from the following papers developed through the years of my PhD. The authors are ordered alphabetically.

Part II:

- Carlo Ghezzi, Claudio Menghi, Amir Molzam Sharifloo, and Paola Spoletini. *On Requirement Verification for Evolving Statecharts*, Requirements Engineering Journal, pages 1-25, 2013.
- Carlo Ghezzi, Claudio Menghi, Amir Molzam Sharifloo, and Paola Spoletini. *On Requirements Verification for Model Refinements*, Requirements Engineering Conference, 2013.
- Carlo Ghezzi, Amir Molzam Sharifloo, Claudio Menghi. *Towards Agile Verification*, Perspectives on the Future of Software Engineering, pages 31-47, 2013.
- Amir Molzam Sharifloo and Paola Spoletini. *LOVER: Light-weight fOrmal Verification of adaptivE systems at Run time*, 9th International Symposium on Formal Aspects of Component Software, pages 170-187, 2012.

Part III:

- Carlo Ghezzi and Amir Molzam Sharifloo. *Model-Based Verification of Quantitative Non-Functional Properties for Software Product Lines*, Elsevier Journal of Information and Software Technology, Volume 55, Issue 3, pages 508-524, 2013.
- Carlo Ghezzi and Amir Molzam Sharifloo. *Dealing with Non-Functional Requirements for Adaptive Systems via Dynamic Software Product-Lines*, Software Engineering for Self-Adaptive Systems II, pages 191-213, 2013.
- Carlo Ghezzi and Amir Sharifloo. *Verifying Non-Functional Properties of Software Product Lines*. Software Product-Line Conference, pages 170-174, 2011.
- Carlo Ghezzi and Amir Molzam Sharifloo. *Quantitative Verification of Non-Functional Requirements with Uncertainty*. International Conference on Dependability and Computer Systems, pages 47-62, 2011.

- Maxime Cordy, Patrick Heymans, Carlo Ghezzi and Amir Molzam Shariflоo, Axel Legay, Pierre-Yves Schobbens. *Formal Modeling and Verification of Non-Functional properties for Software Product Lines*. Technical Report, 2013.[1]

[1]The verification techniques and experiments presented in Chapter 6 originate from this unpublished paper.

Part I

Foundation

CHAPTER 1

Introduction

”The formulation of the problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill.”
Albert Einstein

1.1 Problem Formulation

Modern software is increasingly complex. That is why it is often designed and implemented through an iterative and incremental process. The development process is conducted in a series of iterations, through which a high-level design is refined into detailed specification and finally in machine-readable code. As the result, a large-scale problem is gradually tamed while the evolution is carried out by elaborating and implementing functionalities of interest.

Similarly to iterative-incremental development, the research on adaptive systems deals with some kind of evolution triggered by the need to adapt and modify functionalities at run time [68]. Techniques to develop adaptive systems continuously monitor the changes in the operational environment of a software system in order to adapt their behavior and satisfy the requirements. Adaptation is often realized through introducing some degree of dynamism into the system such that it switches among different configurations depending on the changing environment. There exist different techniques to design and develop *adaptive* systems [77]. In most cases, the ability to adapt is embeded via adding variation points which can be realized by different alternative implementations. For example, a concrete service realizing an interface can be rebound at run time following a service-oriented architecture [37]. This implies that system specification may dynamically evolve over time.

Research on *Software Product Lines* (briefly *SPLs*) is another research direction where variability and evolution play the central roles [21,72]. Relying on planned modularity and reusability, and techniques to introduce variation points and alternatives, engineers derive a large number of different products from a shared SPL specification. In this case, the specification contains all the functionalities while each product takes only a subset.

Although the research on iterative-incremental development, adaptive systems, and SPLs are conducted by different communities, they are linked via the common concern of variability. That is why we take the name of *Variability-Intensive Systems* (briefly *VIS*) to refer to the system specifications produced in these disciplines [1], from which a large number of systems can be derived. In particular, we are interested to guarantee that a VIS specification satisfies its requirements. This is drastically crucial for safety-critical systems, where any violation may lead to expensive penalties. This urges to employ verification techniques able to check specifications before any real implementation and execution. Formal verification, in particular *model checking*, has interestingly progressed through the last two decades and has been been successfully applied in industry [5]. Model checking is an expensive task in terms of time and computation since it exhaustively explores the state-space of specification; however, advanced model checkers (e.g. NuSMV [15]) are able to handle quite large specifications exceeding million states.

Through the thesis, we initially focus on a model-checking problem that has not been properly investigated in the literature. We study the problem of model-checking *incomplete* specifications of system behavior. These specifications are widely produced through early stages of iterative-incremental development, since system specification is gradually refined and documented. This can be also the case in adaptive systems, where some parts of the system are left unspecified at design time. Thus, one of main driving research questions of the thesis is defined as the following:

[1]VISs are further discussed in Chapter 2

RQ.1. *Given an incomplete system specification, how to check whether or not the system behavior satisfies the expected requirements?*

There exist a few works in the literature that address the analysis of incomplete models [31, 75, 88, 89]. In particular, Salay et al. [75] tackle the problem of expressing uncertainty in requirements models. They study this problem for early-stage specifications and propose an approach based on partial modeling technique [31, 76]. Indeed, a proper solution provides suitable formalisms to express incomplete behavior specifications, as one of the basic inputs of a model checker. Accordingly, it is necessary to be able to formally specify incomplete behavior specifications.

RQ.1.1. *How to formally specify an incomplete system behavior?*

According to [75], due to the incompleteness existence the property analysis outputs three values: *True*, *False*, and *Maybe*. This is quite satisfactory for the cases leading to True or False, which refer to property satisfaction and violation respectively. Regarding Maybe, it is essential to describe the condition, in which this result leads to True. Such condition can be viewed as a set of *constraints* defined over unspecified elements. Unlike [75] that mainly deals with First-Order Logic properties, our focus would be on the verification of temporal properties that allows to reason about critical system properties such as safety and liveness [4]. To do that, we need to know how to compute and specify the constraints:

RQ.1.2. *What language can fully specify the set of constraints that lead to the satisfaction of a temporal property?*

RQ.1.3. *How can we generate constraints over unspecified elements of an incomplete specification to guarantee that the original property holds?*

Having the constraints generated, they should be used to compute the verification result when the missing parts of the specification are available. To avoid redundant computations, it is important to avoid re-doing the whole verification from the scratch by only checking whether or not the constraints are satisfied. This is summarized as the following research question.

RQ.1.4. *As the system specification evolves, how can we re-use the re-*

sults of previous verifications to calculate the new results?

Model checking SPLs has been studied in the recent years [18–20, 23]. Most importantly in Classen et al. [19,20], efficient techniques are proposed to concisely represent SPLs and model-check them against LTL and CTL temporal properties [4]. Using these techniques, behaviors of all valid products of an SPL are represented within a supermodel. However, the body of the literature lacks methods to capture stochastic behaviors of SPLs, and similarly the model-checking techniques for stochastic SPLs are missing. Such techniques would allow us to reason about Non-Functional Requirements (e.g. reliability and energy consumption) of all products of a single SPL all at once.

RQ.2. *How to specify and model-check stochastic behaviors of SPLs against non-functional requirements?*

Due to the variety of Non-Functional Requirements (briefly *NFRs*), different formalisms are used to capture behavioral specifications. In this thesis, we focus on reliability and energy consumption as two important NFRs. The existing formalisms for SPLs do not support modeling stochastic behaviors, so further research is needed in this direction. Moreover, we would like to provide both high-level and low-level specifications. The former is used as an abstract means to communicate among different stakeholders, while the latter is the main input to the model-checker for further automatic analysis.

RQ.2.1. *What are the suitable (high-level and low-level) formalisms to capture the stochastic behaviors of SPLs?*

As for each formalism, efficient verification algorithms are demanded to check the satisfaction of NFRs. The existing verification algorithms for stochastic models [52] are mostly limited to single behavior analysis and cannot handle variability of SPLs.

RQ.2.2. *How to efficiently verify non-functional requirements of a SPL?*

As mentioned earlier, variability is the key concept that relates three domains of incomplete system specifications, adaptive systems, and SPLs. Variability may appear in incomplete specifications as unknown system components that can be later elaborated depending upon different design alternatives. We call this kind of variability *open*, since the alternatives are not limited to a predefined set. In terms of SPLs, the alternatives for each variation point are identified and clearly enumerated; and accordingly various products are derived by picking different alternatives . We refer to this kind of variability, in which the alternatives are known, as *closed*. Variability in the context of adaptive systems can be closed or open depending on the adaptation policy. In the former, a certain set of alternatives are specified at design time and conditionally deployed at run time, while in the latter system is allowed to explore and hire new alternatives at run time.

1.2 Contributions

The contributions of this thesis target the motivating research questions on modeling and verification of VISs. We study these problems for both open and closed variability in Parts II and III. Note that Part I gives the necessary background, so it can be skipped if readers are familiar with the preliminary concepts.

Part II focuses on the techniques to verify an incomplete specification against temporal properties, that are appropriate to express safety and liveness requirements. We chose Computational Tree Logic (CTL) as our property language due to its power in expressing temporal properties. However, the general idea of our approach is extensible to other temporal languages e.g. LTL [4]. We motivate the need for verification of temporal properties for incomplete specifications through a running example *Secure Information Retrieval*. We introduce a new formalism *Incompletely Labeled Transition Systems* (*ILTS*), as a new variation of Labeled Transition Systems (LTS), to capture incompleteness. A new verification technique *IMC* is presented to verify ILTS against CTL, which is able to deal with unspecified components. Unlike the standard CTL model checking that is unable to handle unspecified components [2], IMC is able to produce constraints for such components to guarantee the satisfaction of a property. We study the scalability of IMC by running our prototype implementation on the enlarged versions of SIR system. Moreover, we show how IMC can be applied to efficiently check *open* adaptive systems at run time.

Since ILTS formalism is rather low-level, we extend our approach to Statecharts that are well-known to specify system behavior within a user-friendly and compact notation. The new technique is able to verify incomplete Statecharts, and by increasing the verification reusability it alleviates the time to re-verify slightly modified specifications. We discuss the appli-

[2]We use the term *component* as a part of specification that can be plugged and unplugged.

cability of the approach through the classic *Railway Cross* case study, and report on its performance. In this work, *Incompleteness* refers to models in which the inner behavior of some states are unspecified, and are delayed to further refinements. The missing behaviors are abstracted away as states what we later call *Transparent* states. The verification of incomplete Statecharts results in constraints that are indeed *proof obligations* to ensure that initial properties are satisfied.

Part III is dedicated to representing stochastic behaviors of SPLs and verifying them against a set of NFRs, namely reliability and energy consumption. We initially focus on scenario-based specifications, in which augmented Sequence Diagrams linked to feature diagrams and enriched with stochastic information are used to succinctly represent a stochastic SPL behavior. To support the stochastic verification of SPLs, we introduce a new variation of Markov models - FDTMCs - that relies on the notion of features as the first-class concept. We describe three different techniques to verify FDTMCs against two stochastic temporal logics: PCTL and Reward logic, that are popular to express reliability and energy consumption properties. We study efficiency of the approaches through two case studies. Later, we investigate the verification of non-functional properties for *closed* adaptive systems, and present an approach based on *Dynamic* SPLs.

1.3 Thesis Organization

The remainder of the thesis is structured as below. Chapter 2 gives a brief introduction to VISs. Chapter 3 provides an overview on the fundamental concepts of model checking. Chapter 4 motivates the notion of incomplete model checking and describes the verification of CTL properties in such setting. Chapter 5 extends this approach to Statecharts and presents a methodology to incrementally design and verify state-based specifications. Chapter 6 discusses stochastic formalisms and verification techniques for SPLs, and Chapter 7 presents a dynamic model for SPLs to build and verify adaptive systems. Finally, Chapter 8 concludes the thesis by explaining how our research questions are addressed, and discusses the current limitations and future work.

CHAPTER 2

Variability-Intensive Systems

"Nothing endures but change." Heraclitus

Evolution is the key in software engineering through both system design and maintenance. It is quite common that the initial design provides a big picture of system upon which stakeholders reach a common understanding and communicate requirements and constraints. Such high-level description can be viewed as the system skeleton, which may be referred as architecture [86] or system metaphor among developers. The initial description is further elaborated and may be modified as the implementation is progressed, and meanwhile different design choices are applied and their impacts are evaluated. The inevitable fact is that many changes may be applied to specifications, which may lead to unforeseen defects and can diminish the quality of both specification and implementation.

A *modular* design allows developers to isolate the impact of changes and to prevent the propagation of local changes to the whole system. Modular software construction is supported by a variety of methods and techniques, e.g. *component-based software engineering* [45], *feature-oriented software engineering* [3], *aspect-oriented programming* [27]. Modular specifications allow us to specify a large system though composing modules, which interact with each other and are hierarchically decomposed into fine-grain modules. Following such hierarchical and compositional approach not only simplifies implementation, testing, and project management, but also enables the possibility to outsource system components and rely on external services offered by third parties.

Identifying and managing variability are of most important to tackle evolution. Through a modular design, invariant and variable modules can be identified and production activities can be accordingly customized such that variability would be effectively managed. Consider a system comprising three components A, B, and C. Components A and B are reutilized from a legacy system, and so they are well-tested. Component C is still abstract and is to provide a new functionality for users. There might be different design/technology alternatives to build component C, which requires a developer to carefully explore their pros and cons, and select the one that better fits the requirements and global system expectations. Modularity gives us the capability to highly focus on the uncertain parts of systems, which might vary. When the alternative choice of a variable module is selected, analyzed and stabilized, then it is considered as invariant. This way a system can be hierarchically and incrementally designed, elaborated, and implemented such that the variability is properly viewed and a design rational is settled. This also alleviates any efforts for future changes, in case it turns out that a design decision is not effective as expected. Modeling variability has been studied within software engineering community in the past years. In this regard, there have been dedicated workshops such as VAMOS [1] and conferences e.g. SPLC [2], which are the meeting points for researches to present and discuss their innovations.

In this thesis, we consider three kinds of specifications and refer to them as *Variabilty-Intensive Systems* (VIS), in which variability is treated as a first-citizen concept: *Incomplete Specifications*, *Software Product Lines* (SPLs), and *Adaptive Systems*. Incomplete specifications are those in which some system modules are unspecified. This kind of specifications are the intermediate artifacts at design time. Unlike incomplete specifications, in which variation points are identified but not alternatives, SPLs provide alternatives for each *variation point* as well. Variation point is the term used to refer to a module or attribute of system that is subject to vary. SPLs are able to represent a large number of variation points and alternatives, and are of special interest by industry. Unlike incomplete specifications and

[1]International Workshop on Variability Modelling of Software-intensive Systems

[2]Software Product Line Conference

SPLs that resolve variability at design time, adaptive systems leave variation points to run time to support dynamic binding and to enable a system to dynamically modify its behavior on-the-fly. The rest of this chapter briefly introduces the three kinds of VISs.

2.1 Incomplete Specifications

Incomplete Specification refers to intermediate system models that are developed during development process, and demonstrate how a system behaves. To tame the boosting complexity of modern systems, derived from the variety of features requested by customers and increasing number of design and technology choices, developers apply iterative process models through which a system is incrementally designed and implemented. This is done by starting from a high-level abstract model in which the behavior of some components are not well-known, which will be elaborated and captured later as the design is progressed. We refer to these intermediate abstract models as *incomplete specifications*. An incomplete specification is completed as soon as the behavior of unspecified components are provided.

In the literature, *partial models* also refer to usage scenarios of a software system. Through a scenario execution, system may perform different operations while interacting with users and its environment. They are called partial because each scenario demonstrates only some of the behaviors that a system may exhibit [33]. Chechik et al. [31, 75] discusses uncertainty at early requirements modeling and annotate some elements of models with Maybe tags, referring to the fact they may or may not exist in the model. We show two examples of incomplete specifications in Sections 4 and 5.

2.2 Adaptive Systems

Software maintenance is a response to the needs that a released software system cannot properly address a variety of users requirements. Traditionally software maintenance is considered a manual task that is performed through human intervention. However, software can be found everywhere, in many devices and a variety of domains. It is economically unreasonable to keep the maintenance as a manual human work. There are cases that human intervention is not even possible e.g. repairing autonomous robots in space exploration projects. Hence, many maintenance techniques have been pushed to software from human; load balancing and fault tolerance are two examples.

Developing self-adaptive systems is a new approach to dealing with the challenges raised by software maintenance. "*Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation.*" [68]. Accordingly, the main characteristic of a self-adaptive system is that the system is continuously looking for any change in both its operating environment and its internal elements, and is actively responding to them in order to satisfy its main goals [58].

The adaptation process is often implemented through a closed loop in which system observes the environment, collects and analyzes data and diagnoses violations from requirements and accordingly plans to prevent from them. This process resembles the IBM's vision of autonomic computing [50]. Adaptations are either architectural or parametric [64]. While the former deals with system-level changes, e.g. component replacement, the latter deals with tuning system parameters, e.g. database pool size, in order to full requirements and reach a higher efficiency. Adaptation planning may be online or offline. Offline plans are static adaptation scripts provided at design time and mainly based on a priori knowledge. On the contrary, online planning applies learning techniques by using collected run-time data to generate and update plans at run time.

Through the thesis, we only focus on component-based adaptations in which some parts of systems may change and the rest of the system remains invariant. Our techniques don't touch adaptation planning so it can be performed in any preferred manner.

2.3 Software Product Lines

"A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a particular market or mission, and that are developed from a common set of core assets in a prescribed way" [21]. A development approach inspired by SPL principles aims at improving productivity and reducing the time and cost needed to develop a family of products. SPL engineering attempts to produce a large number of products that share a set common features in a systematic manner such that each specific product is tailored for a category of users.

Feature is a basic concept in developing SPLs. Apel and Kastner [3] refer to a feature as a *unit of functionality* that satisfies a requirement. *Feature Diagrams* (FDs) are a visual formalism to capture features and their relationships of a SPL, and we will use them in the second part of the thesis to model SPL variability. FDs are used to specify the valid products that can be derived from a SPL. In principle, they are trees in which features are the nodes, shown by rectangles, and the relationships are actually the decomposition of features depicted by edges. Given a parent feature, a decomposition edge constrains the combination of the child features. Figure 2.1 shows the FD for the vending machine that we further discuss in Chapter 6. Accordingly, the vending machine is decomposed into three features - *Beverage*, *Payment*, and *Taste* - by using AND operator. AND decomposition imposes the existence of the child features, unless they are *optional*. Taste is an optional feature which is indicated by adding a circle in the diagram. On the contrary, Beverage and Payment are *mandatory* features. A feature may be decomposed by OR operators, which are either *inclusive* or *exclusive*. Inclusive OR - shown as OR - are the case in which at least one of the child features are included in a product, while exclusive OR - shown as XOR - requires that only one child feature is allowed in a valid product.

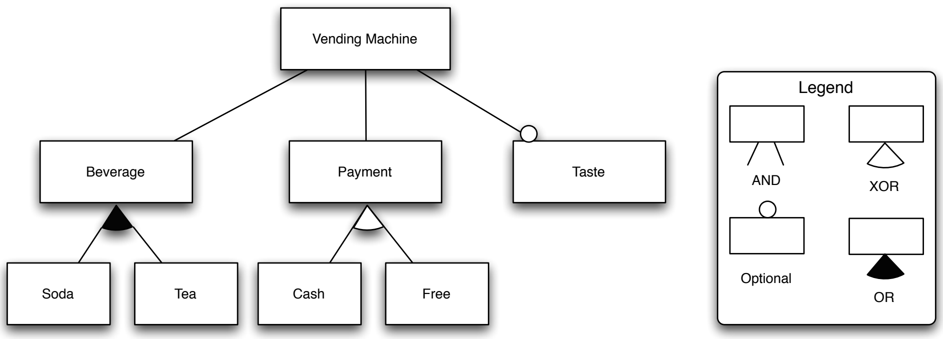


Figure 2.1: *The vending machine's FD*

Some variation of FDs provide cross-cutting relations - requires and excludes -, that are useful to model feature dependencies. The presence of a feature may require the presence of another feature in any valid product. Similarly two features may have conflicting natures, and their inclusion together may be shown by *excludes* relation.

FDs are nice formalisms to capture and visualize variability and commonality of SPLs; however, they provide no information about behavior. To fill this gap, behavioral formalisms have been proposed for SPLs. FTSs [20] are a variation of Transition Systems that are proposed to capture behaviors of all products of a SPL via a compact representation. FTS's transitions are annotated with *feature expressions*, which are boolean expressions used to enable and disable transitions. In order to derive each product, the boolean expressions are evaluated and depending on the value, the transitions evaluated false are eliminated from the resulting TS of a product.

Classen et al. [19, 20] presents efficient algorithms for model checking LTL and CTL properties of all products of SPL at once; however, FTSs are expressive enough to capture stochastic behaviors of SPLs. In chapter 6, we discuss behavioral modeling of SPLs and provide formalisms and later verification techniques to deal with variability.

CHAPTER 3

Formal Verification

"*If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.*" John Louis von Neumann

Software systems have been pervading every aspect of the human life. Society became totally dependent on software, both in terms of the functionalities it supports and of their quality, which may ultimately affect their usefulness. It is thus crucial that we can guarantee assurance that a given software satisfies a set of predefined properties, which represent the functional and non-functional *requirements* the system must fulfill. Functional requirements concern the effect of operations the system is expected to deliver whereas non-functional requirements concern their qualities, such as performance, availability, usability, energy consumption, and cost. Software *verification* aims at ensuring that a system executes according to some specified desirable functional and non-functional behavior. Verification is a most important activity performed during software development and evolution. In practice, it is normally achieved by *testing* [71], i.e., by sampling a representative set of behaviors that are deemed to provide useful information about the running conditions that will be encountered by the software when it will be operational. *Formal verification*, instead, aims at mathematically proving that given properties, which specify the desired requirements, are indeed satisfied by the system.

Model checking is recognized as a successful kind of formal verification. Given a system model $\mathcal{M}$ and a formal property ϕ, model checking systematically and exhaustively checks whether ϕ holds for $\mathcal{M}$ [4]. The model may be an abstraction generated from code, e.g. C or Java; or it may be a high-level specification that is developed during design to support some reasoning about the system under construction (the *system-to-be*). It represents the system's behavior in an abstract, yet precise and non-ambiguous, mathematical form. The property specifies instead the requirements the system must satisfy. The overall idea behind model checking is to explore the state space of the model, and ensure that the properties of interest are satisfied by considering all possible behaviors.

Formal verification has now become mature. It has already been used in practice in several application domains and has been adopted in several industrial settings [5]. In particular, model-checking techniques have been substantially improved over the years, and can in principle complement testing to achieve improved assurance. Figure 3.1 shows the classic model-checking process, in which system specification and the property of interest are fed to a model checker. The result of model checking is *True*, in case the property is satisfied by the specification, or *False*, when the property doesn't hold. In the latter case, counterexamples are generated to provide examples of traces that lead to the violation.

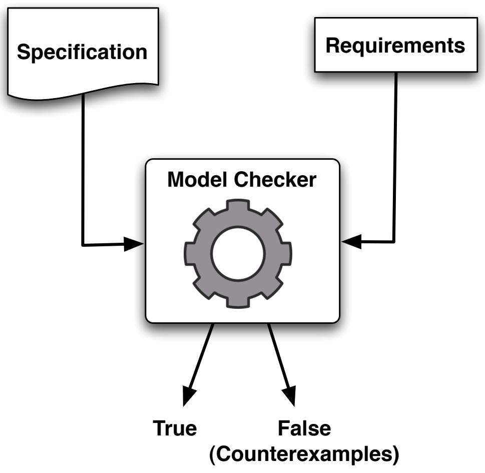


Figure 3.1: *Model Checking Process*

Model-checking techniques highly depend on the type of properties to be checked. The remainder of this chapter briefly provides the basic concepts existing in the literature upon which the proposed techniques are built. System specification is an abstract model that represents system behavior comprising the information that is required to check properties. Depending on the system characteristics, different specification languages may be chosen. Section 3.1 gives a brief discussion on formal specifications that are extended for variability-intensive system in the successive chapters. We discuss languages used to formally express temporal properties, to specify requirements, in Section 3.2. Finally the basic concepts behind model checking of temporal properties is given in Section 3.3

3.1 Formal Specification

System specification is an important input of a model-checking task that abstracts away irrelevant details and represents the information served to check the properties of interest. Behavioral specifications provide an abstract model of system that represents how system operates and interacts with its environment. Transition systems are one of the common modeling formalisms that are often used in computer science. They nicely show a system behavior in terms of states and transitions, which together demonstrate how system evolves through the occurrences of actions. In the following, we initially give the formal definition of transition systems, and then present Markov chains as the extension capable of representing probabilistic behaviors.

3.1.1 Transition Systems

A *transition system* is defined as a tuple $\langle S, s_0, Act, \rightarrow, L\rangle$ over atomic proposition AP where

- S is a set of states;
- s_0 is the initial state;
- Act is a set of actions;
- $\rightarrow \subseteq S \times Act \times S$ represents the transitions between states;

- $L : S \rightarrow \wp(A)$ is a labeling function;

Labeling function L associates each state with a set of atomic propositions, which are true in that state. Each state represents the state of a system at a certain point through the execution. The system is initially in state s_0, and switches to a new state by taking one of the outgoing transitions and performing its action. Here we consider only one initial state, but it can be a set of states. In case, there are more than one outgoing transition, one of them is non-deterministically chosen. The execution is followed by visiting states and taking transitions, and is terminated when a state with no outgoing transition is reached. Actions and propositions out of interest are not captured, this is why sometimes a transition system has no action or is labeled only with a few propositions. By removing the actions from TS, a new formalism called as Labeled Transition Systems (LTS) is defined. An LTS is a quadruple tuple $\langle S, s_0, \rightarrow, L\rangle$ over atomic proposition AP where

- S is a set of states;

- s_0 is the initial state;

- $\rightarrow \subseteq S \times S$ represents the transitions between states;

- $L : S \rightarrow \wp(A)$ is a labeling function;

Figure 3.2 shows an LTS for a simple traffic light. There are three states labeled with G, Y, and R stand for Green, Yellow, and Red respectively. The initial state is G, and actions which are to turn off and turn on different lights are omitted for simplicity. Using temporal logic[1], we can intuitively express some properties that hold for this traffic light. For example, $GFgreen$ is a liveness property that is true saying that there is always a possibility to show green light.

3.1.2 Probabilistic Models

Probabilistic models provide a very expressive power to specify uncertain and unpredictable behaviors in a quantitative manner. Using randomization in distributed protocols of computer networks is one of the examples

[1]Temporal Logic is briefly presented in Section 3.2.

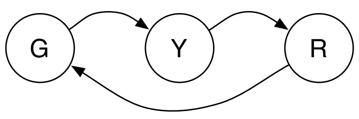


Figure 3.2: *Transition system for a traffic light*

leading to unpredictable behaviors. Other examples of stochasticity are in the description of user request submission rates in interactive environments. Probabilistic model checking has been developed in the recent past to verify models that exhibit stochastic behaviors. It has been used in various domains from biological systems to sensor networks. It can be also used for verifying and predicting non-functional properties of software systems, such as reliability, performance, and various kinds of "costs", including energy consumption of computational functions. The underlying models used in probabilistic model checking are different kinds of Markov models, including Discrete-Time Markov Chains (DTMCs), Continuous-Time Markov Chains (CTMCs), Markov Reward models, etc. Recently, there have been great improvements in the tools and techniques for probabilistic model checking. PRISM is a widely used probabilistic model checker that is currently used both in research and in industrial settings [4, 52].

DTMCs are LTLs with augmented probabilities over transitions, where the sum of probabilities on outgoing transitions of every states equal to one. This is a way to deal with non-determinism among outgoing transitions. DTMCs are widely used to represent and reason on reliability aspects of software systems. Figure 3.3 shows a DTMC to model the probabilistic failure of a traffic light. The DTMC is constructed upon the transition system in Figure 3.2, by adding state F, in which the traffic light is failed. Accordingly, each time the color of traffic light is changed with a probability of 0.999, the system may fail with the probability of 0.001. The traffic light keeps working until it terminates in state F.

DTMCs with Rewards also label transitions with real numbers representing rewards or costs of a transition (which may, for example, model the energy consumption or execution time of actions). Rewards may be associated with states, which has the same expressivity of assigning them to transitions and can be simply transformed to such DTMC. In this case, they represent the cost/reward of remaining in a state. Figure 3.4 shows the DTMC with Reward for traffic light example, in which states are associated with execution time. This model can be used to reason about the time spent in different states. Similarly this value can represent the average en-

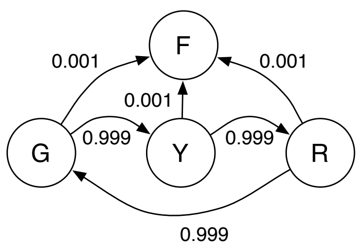


Figure 3.3: *DTMC for modeling failure in traffic light example*

ergy consumed in each state, so it allows us to calculate the average energy consumption given a certain number of steps.

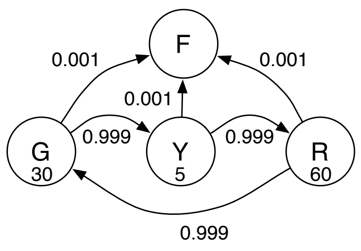


Figure 3.4: *DTMC with Rewards for traffic light example*

3.2 Formal Languages for Requirements Specification

Requirements are initially expressed in natural language or informal models through requirements elicitation phase. That is to communicate them among different stakeholders including users, developers and managers. However, requirements are specified formally to avoid ambiguity in natural language, and to allow the application of automatic tools for further analysis. The choice of a language mainly depends on the type of requirements. For example, if requirements are to specify a functionality in terms of pre-condition and post-conditions, then boolean formulae and first-order logic might be convenient choices. However, if the properties constrain system execution and traces, then temporal logics may come to the scene.

Properties like safety, liveness, and robustness can conveniently be expressed using temporal logics such as LTL and CTL. LTL is designated for properties over a linear time, whereas CTL is based on branching time. Unlike the linear time, CTL allows to describe properties over unpredictable environments, where different operations may occur given a state. Our focus in this thesis is on CTL for classic temporal properties and its probabilistic variation, PCTL. The reason behind this choice is the existence of simpler and more intuitive model-checking algorithms for branching time logics.

Quantitative temporal logics e.g. Reward Logic [52] are good to formally express non-functional properties like cost and energy consumption of a given behavior, while probabilistic temporal logics like PCTL and CSL are suitable for reliability and performance, respectively. We will show the applicability of these logics to constrain non-functional requirements of SPLs.

Since specifying requirements in a formal form requires a background in logic, which might go beyond the expertise of many engineers, a couple of pattern systems have been proposed [9,25,39]. For example, Grunske [39] presents a pattern system ProProST and tool support to map a structured natural language grammar to probabilistic temporal logic CSL. Using ProProST, requirements can be written in a structured form of natural language, while the formal specification is automatically produced by the tool.

In the below, we present the formal logics that are used to express requirements through the chapters of this thesis.

3.2.1 Computational Tree Logic

CTL is a branching-time logic in which the model of time is a non-deterministic tree, leading to different paths [16]. It can be used to specify different properties on all or some of program executions. For example, a safety requirement may be expressed by constraining that an *error* state is unreachable by any execution paths. Quantifiers *All* and *Exists* are added to specify properties over paths, which are shortened as A and E resepectively.

The formal syntax of CTL is as below.

$$\begin{aligned}&\Phi ::= true \mid a \mid \Phi \ \wedge \ \Phi \mid \neg\, \Phi \mid \ E\Psi \mid A\Psi\\&\Psi ::= X\Phi \mid \Phi U \Phi\end{aligned}$$

where Ψ and Φ stand for *path formulae* and *state formulae*, respectively. Every atomic proposition $a \in AP$ is a state formula and $\wedge$ and $\neg$ are the usual boolean operators [4].

The semantics of CTL is defined on a *Transition System*, where states are labeled with atomic propositions or their boolean combinations. A path is defined as a subsequence of states. X stands for the *next* path operator. $AX\Phi$ intuitively means Φ holds in all successor states, while $EX\Phi$ holds when Φ holds at least in one of successor states. Formula $\Phi_1 U \Phi_2$ holds for a path if there is some state along the path in which Φ_2 holds and Φ_1 holds in the preceding states. Consequently, $A\Phi_1 U \Phi_2$ holds if $\Phi_1 U \Phi_2$ holds for all paths, whereas to satisfy $E\Phi_1 U \Phi_2$, it is enough if there exists only one path for which $\Phi_1 U \Phi_2$ holds. Other operators like *Always* and *Eventually* can be derived from Until [16].

The safety requirement mentioned earlier can be expressed as $\neg EFerror$, where F stands for Eventually, and it means that there is no path that reaches the state labeled with *error*.

3.2.2 Probabilistic Computational Tree Logic

Sometimes it is not possible to absolutely guarantee the correctness of a system property, e.g. "connection never fails". There are real-world system properties e.g. reliability that are expressed in a quantitative way. A car rental company may guarantee its service in a quantitative way due to unforeseen problems. For example, the car arrives on time with a probability greater than 0.95. An Internet Service Provide offers a very cheap Internet tariff with a probability of 0.1 connection failure per hour and an expensive tariff with a probability of only 0.0001 connection failure per hour.

Probability theory is a common way to dealing with such uncertainty. Markov chains are the most popular operational models to capture probabilistic behaviors to analyze reliability and dependability of software systems. Discrete-Time Markov Chains (DTMCs) are transition systems with probabilities over transitions, where the sum of probabilities on outgoing transitions of every state equals to one. This is a way to deal with non-determinism among outgoing transitions. DTMCs are formalism that are nicely applied to reason about reliability models of software systems. There are variations of temporal logics to express probabilistic properties. Probabilistic CTL, hereafter PCTL, is a variation of CTL that uses probability operator instead of path quantifiers. PCTL is defined by the following syntax:

$$\begin{aligned} \Phi &::= true \mid a \mid \Phi \ \wedge \ \Phi \mid \neg \ \Phi \mid \mathcal{P}_{\bowtie p}\left(\Psi\right) \\ \Psi &::= X\Phi \mid \Phi U^{\leq t}\Phi \end{aligned}$$

where $p \in [0,1]$, $\bowtie \in \{<, \leq, >, \geq\}$, $t \in \mathcal{N} \cup \{\infty\}$, and a represents an atomic proposition. Similarly to CTL, the temporal operator X is called *Next* and U is called *Until*.

The satisfaction relation for PCTL is defined for a state s as:

$$\begin{array}{l}
s \models true \\
s \models a \quad \text{iff} \quad a \in L(s) \\
s \models \neg\Phi \quad \text{iff} \quad s \not\models \Phi \\
s \models \Phi_1 \wedge \Phi_2 \quad \text{iff} \quad s \models \Phi_1 \; and \; s \models \Phi_2 \\
s \models \mathcal{P}_{\bowtie p}(\Psi) \quad \text{iff} \quad Pr(s \models \Psi) \bowtie p
\end{array}$$

A complete formal definition of $Pr(s \models \Psi)$ can be found in [4]; details are omitted here for simplicity. Intuitively, its value is the probability of the set of paths starting in s and satisfying Ψ. Given a path π, we denote its i-th state as $\pi[i]$; $\pi[0]$ is the initial state of the path. The satisfaction relation for a path formula with respect to a path π originating in s ($\pi[0] = s$) is defined as:

$$\begin{array}{l}
\pi \models X\Phi \quad \text{iff} \quad \pi[1] \models \Phi \\
\pi \models \Phi_1 U^{\leq t} \Phi_2 \quad \text{iff} \quad \exists 0 \leq j \leq t.(\pi[j] \models \Phi_2 \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi_1))
\end{array}$$

From the *Next* and *Until* operators it is possible to derive others. For example, the *Eventually* operator (often represented by the F) is defined as:

$$F^{\leq t}\phi \;\equiv\; true\, U^{\leq t}\phi$$

It is customary to abbreviate $U^{\leq\infty}$ and $F^{\leq\infty}$ as U and F, respectively.

PCTL can naturally represent reliability-related properties for a DTMC model of the application. For example, we may easily express constraints that must be satisfied concerning the probability of reaching absorbing failure or success states from a given initial state. These properties belong to the general class of *reachability properties*. Reachability properties are expressed as $\mathcal{P}_{\bowtie p}(F\ \Phi)$, which expresses the fact that the probability of reaching any state satisfying Φ has to be in the interval defined by constraint $\bowtie p$.

3.2.3 Reward Temporal Logic

While PCTL is much suitable to constrain the reachability probability over paths, it is not expressive to specify properties that deal with rewards and costs obtained/spent by execution a transition or visiting a state [52]. Indeed a path can represent the execution of a sequence of operations, and some interesting non-functional requirements may be defined over the total reward/cost collected by firing a set of transitions. For example, one may define a requirement to constrain the maximum energy consumption used by system to reach a specific state. This can be particularly of interest for energy-restricted devices like smart-phones. Reward logic allows to specify properties over the reward/cost attached to transitions or states of DTMCs.

Reward logic uses one of the following patterns, in which R is the reward operator. $\sim r$ represents a comparison relation ($\leq r$, $\geq r$, and $= r$) in which r is a real number.

$$R_{\sim r}[C^{\leq t}] \mid R_{\sim r}[F\phi] \mid R_{\sim r}[S]$$

$R_{\sim r}[C^{\leq t}]$ is called a *cumulative reward property* and corresponds to the cumulated reward along a path. However, the path is limited by t which is the path length. The accumulation manner leads summing the rewards over the path.

$R_{\sim r}[F\phi]$ represents a reachability property for a path starting from an initial state and ending in states where ϕ holds. ϕ is a state formula in this case.

$R_{\sim r}[S]$ is to verify steady-state properties. In fact, this property is not related to paths but rather to the long-run of system. The semantics is used to constraint the average reward accumulated by starting from state S in the long-run.

3.3 Model Checking Techniques

The research on model checking of temporal properties has been active through the recent decades. The advent of efficient model-checking techniques beside the continuous advances in hardware technologies make their application to real-world cases more feasible last decades.

Model checking exhaustively searches the state space of system behavior to check whether or not a certain property holds. Counterexamples may be reported as the output in the case the property is not satisfied. There exists a variety of model checking algorithms for different temporal logic languages. For example, there are basic model checking algorithms for LTL and CTL, that are well-known for years. These algorithms are usually based on explicit representation of state space. Symbolic model checking is an attempt to handle large systems by applying efficient data structures. In this regard, Binary Decision Diagrams (BDDs) [10] have been of most interest to represent objects such as transition systems and system states. There exist symbolic model checkers for different temporal logics, e.g. NuSMV [15].

In the sequel, we overview the basics of model-checking approaches for CTL and PCTL, which are later referred in the successive chapters.

3.3.1 CTL Model Checking

Given a transition system TS and a property ϕ the mode-checking problem for CTL is to check whether or not $TS \models \phi$. CTL model checking can be performed through a recursive procedure that calculates $Sat(\phi)$, which stands for the satisfaction set of CTL sub-formulae of ϕ. The property holds if and only $I \subseteq Sat(\phi)$, which means if all initial states belong to the satisfaction set of ϕ.

To compute $Sat(\phi)$, the parse tree of ϕ is constructed and the satisfaction set of each node is calculated through a bottom-up approach. While the leaves of parse tree are propositions, the inner nodes are boolean and temporal operators. Initially the satisfactory states of each leave is computed by checking the satisfaction of its propositions. The satisfaction set of inner nodes are calculated with obtaining the satisfaction set of children, each of which is a sub-formula of ϕ, and applying functions that follow the semantics of their operators.

Algorithm 1 presents the recursive evaluation to compute the satisfaction set of ϕ. It takes as inputs a subtree T of the parsing tree, formula φ, and LTS M on which φ is evaluated. T is a binary tree, where a node representing a unary operator has a single child, while a node representing a binary operator has two children. We use $T.S$ to refer to the set of states in M that satisfy the formula represented by the current subtree, $T.left$ and $T.right$ to refer to the left and the right subtrees of the current tree (when the root is a binary operator), and $T.son$ to refer to the subtree of the current tree (when the root is a unary operator). The elements of M are referred as $M.S$ and $M.L$ (labeling function). The set X (initialized in line 2) is used as a local set to store the elements that satisfy φ.

The boolean *and* leads to the intersection set of the satisfaction states of the children nodes. As for Until operator ($\phi_1 \cup \phi_2$), the satisfaction states of ϕ_2 are added as well as all the states of $Sat(\phi_1)$ that make a path leading to a state in which ϕ_2 holds. Finally, Always operator returns all the states of the only child node which also belong to at least one Strongly Connected Component (SCC) of TS.

3.3.2 PCTL Model Checking

Given a DTMC model M and a PCTL property ϕ, the model checking problem of ϕ is to check whether the initial states I is a subset of $Sat(\phi)$. Model

Algorithm 1 Recursive calculation of the satisfaction set

```
evaluate($\varphi, T, M$){
  $X = \varnothing$;
  switch ($\varphi$){
    case $true$ : $X = M.S$;
    case $\varphi \in AP$ :
      for all $s \in M.S$
        if ($AP \in L(s)$)
          $X = X \cup \{s\}$;
    case $\varphi = \varphi_1 \wedge \varphi_2$ :
      for all $s \in Sat(\varphi_1) \cap Sat(\varphi_2)$
          $X = X \cup \{s\}$;
    case $\varphi = E\varphi_1 U \varphi_2$ :
      $X = Sat(\varphi_2)$;
      $X' = \varnothing$;
      while($X'! = X$){
        $X' = X$;
        for all $s_1 \in T.left.S$
          if($\exists s' \in X | (s_1, s') \in M.Transitions$)
            $X = X \cup \{s_1\}$
   case $\varphi = EG\varphi_1$ :
    for all $sub_S \in Scc(T.son.S)$
        $X = X \cup sub_S$;
  $T.S = X$;
}
```

Checking of PCTL properties follows a bottom-up approach similarly to CTL model checking to calculate the set of satisfaction states; however, the probabilistic operator P_J needs to be treated differently. Given initial state s, $s \models \phi$ holds if the calculated probability is in the bound indicated by J.

$$Sat(P_J(\phi)) = \{s \in S | Pr(s \models \phi) \in J\} \tag{3.1}$$

To compute $Pr(s \models \phi)$, we need to provide procedures over path operators Next and Until. Regarding Next operator, the probability is computed as the sum of probabilities over transitions that lead to successive state s', in which ϕ holds: $\sum_{s' \in Sat(\phi)} P(s, s')$ where P is the probability matrix of DTMC M.

The treatment of Next operator is extended to deal with bounded Until, in which the length of the path is limited with the bound n. The reachability probability of state s within the bound n is obtained by P^n that stands for n times of matrix multiplication of P. The case of unbounded Until requires solving a linear equation system. The interested reader is referred to [4] for an explanatory discussion.

Part II

Modeling and Verification of Open Variability

CHAPTER 4

Verification of Incomplete Specifications

"To demonstrate that all human knowledge is incomplete and all human truth partial is not to demonstrate that all human knowledge is ignorance and all human truth false or some ambiguous thing between true and false."
David Potter

4.1 Introduction

System specification is the key means to understand how a system operates and how it interacts with its environment. As an abstraction of the real implementation, by avoiding unnecessary details, it is useful to reason and predict system behaviors. As a consensus, specification analysis techniques mostly depend on the fact that the specification is complete; however, there are many cases through software development process in which specification might be incomplete but still some kind of reasoning and quality assurance would be beneficial. Specification-based reasoning is of importance in model-driven software engineering, where the development starts with abstract models and the implementation is generated through incremental refinements. As a matter of fact, earlier detection of design flaws

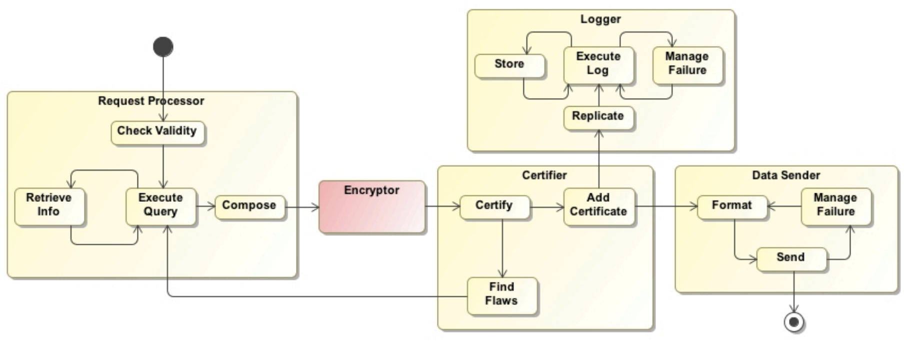


Figure 4.1: *The activity flow of Secure Information Retrieval*

leads to cheaper corrections. Thus, analyzing specifications and checking the satisfaction of requirements play a key role in reducing maintenance cost.

Beside the ability to check incomplete specifications against system requirements, it is also important to reuse the results of previous analysis to speed up the whole procedure. For example, some parts of a specification may change frequently, because the design team is unsure about possible alternatives, while the rest may remain invariant.

Let us consider the following case as an example of incomplete specification. *Secure Information Retrieval* (SIR) is an information system that receives requests, in form of questions, from the clients and responds to them via encrypted messages. The system behavior, in terms of the interactions among the components, is illustrated in Figure 4.1.

A request received from a client is processed by *Request Processor* component. First, the validity of the request is checked and then the requested information is retrieved by querying on different data centers. The results are composed as a message to be sent to the client. This message is encrypted by an *Encryptor* component. The encrypted message is checked against a set of security standards by a *Certifier* component. The certified message is logged and sent to the client. For security and reliability reasons, the following set of properties shall be guaranteed by the system.

Security property: any message shall be encrypted before being sent out over the network;

Reliability property: the system shall recover from any failure.

As illustrated in Figure 4.1, Encryptor component is shown with a different color, since the designer is not able to make a final decision about its realization, and so this decision is postponed to next development stages. It can be also the case that some components are bound and frequently rebound at run time, which usually occurs in adaptive systems. However, it is useful to check weather or not the design, even though incomplete, has the potential to satisfy the properties. It can be the case that the design violates the properties regardless of missing decisions. In this case, whatever component we plug to the system, the properties are violated. The situation can be completely different, where the design always satisfies the properties regardless of the missing decisions. However, in many cases such satisfaction depends on the missing parts of the design specification.

Our goal is to take the initial steps to address the problem of verifying incomplete specifications. We first briefly present the model checking process of an incomplete specification. Then we realize this process by focusing on CTL properties.

4.2 Incomplete Model Checking (IMC)

The conventional model checking takes two inputs: *specification* and *requirements*. Specification represents the system under study, while the requirements are the properties that system is supposed to satisfy. Given each property of interest, the model checker explores the state space of the specification and checks whether or not the property holds. Although specification is an abstract model of system and does not contain many details but it is complete in the abstraction level and contains the necessary information for the model-checking purpose.

Unlike conventional model-checking approaches, IMC deals with incomplete models, where some parts of the system are unspecified. IMC allows the designer to verify the incomplete system specification and to calculate a set of constraints for the unspecified parts. These constraints guide further design decisions to avoid deviating from global system properties. Figure 4.2 shows IMC as well as conventional MC.

Although IMC can be viewed as a general process to verify incomplete specification against a variety of properties, our focus is on temporal properties expressed in CTL. In the next section, we introduce a variation of LTS to capture incomplete specifications. Then, we provide a formal definition for the logics that are used to express properties. Later, we present a model-checking algorithm to verify such incomplete specifications against CTL properties.

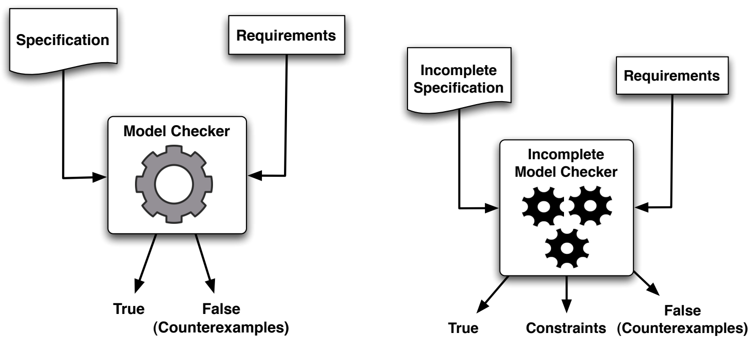


Figure 4.2: *Two model checking styles: Conventional (Left) versus Incomplete (Right)*

4.2.1 Incompletely Labeled Transition System

An Incompletely Labeled Transition System (ILTS) is a Labelled Transition System (LTS) in which the set of states is partitioned in R, the set of *regular* states, and T, the set of *transparent* states, that are special states that can represent more complex components and are considered as unknown. Formally, an ILTS is specified as a tuple $\langle S, s_0, \rightarrow, L\rangle$ over the alphabet AP of atomic propositions, where

- S is a set of states, which is partitioned in two sets: R (Regular) and T (Transparent) , i.e., $S = R \cup T$ and $R \cap T = \varnothing$;

- s_0 is the initial state;

- $\rightarrow \subseteq S \times S$ represents the transitions between states;

- $L : R \rightarrow \wp(AP)$ is the labeling function that associates a subset of atomic propositions to each regular state.

The transparent states represent unknown components that, once specified, can be modeled using a special kind of LTS, namely LTS with single final state, i.e., a tuple $\langle S, s_0, s_F, \rightarrow, L\rangle$, where $s_F \in S$ is the final state. The initial and final states represent the unique entry and exit points in and from the component, respectively. For simplicity, our ILTS excludes actions; instead, in the next chapter we discuss incompleteness in Statecharts which not only include actions but also hierarchy and concurrency together.

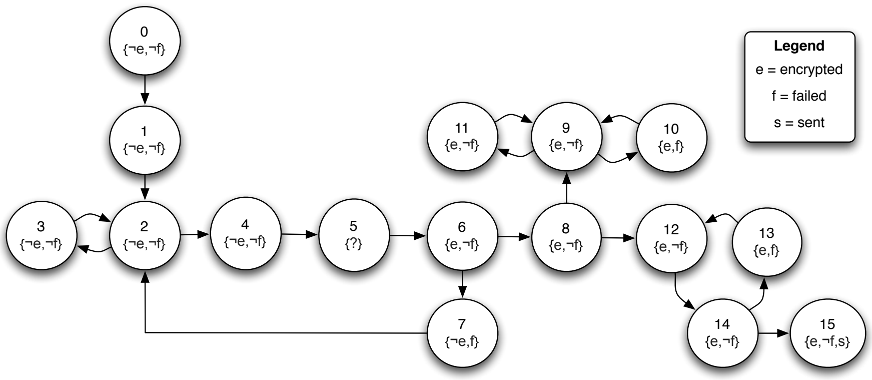


Figure 4.3: *The ILTS of the Secure Information Retrieval system*

Figure 4.3 shows the ILTS of SIR system, which is driven from the activity flow, shown in Figure 4.1. Transparent state 5 represents the unavailable specification of Encryptor. The other states are labeled regarding the three message attributes: encrypted, failed, and sent.

4.2.2 Next-Free CTL and Path-CTL

We define Next-Free CTL as a proper subset of CTL that excludes Next operator. Hence, the syntax of the language becomes:

$$\phi \rightarrow \ \phi \wedge \phi \mid \neg\phi \mid E\ \phi\ U\ \phi \mid E\ G\ \phi \mid p$$

where $p \in AP$, EU and EG are the CTL operators whose semantics is briefly recalled below. Comparing to the definition in chapter 3, we eliminated the quantifier A, because it can be simply obtained by $\neg E \neg$.

Let's recall the semantics of Next-Free CTL on a state of LTS $M = \langle S, s_0, L\rangle$ ($M,\ s \models\ \varphi$ means that φ holds in a state s of the LTS M) as follows:

- $M,\ s \models\ p\ \Leftrightarrow\ p \in L(s)$;
- $M,\ s \models\ \neg\varphi\ \Leftrightarrow\ M,\ s \nvDash \varphi$
- $M,\ s \models \varphi_1 \wedge \varphi_2\ \Leftrightarrow\ M,\ s \models \varphi_1$ and $M,\ s \models\ \varphi_2$;
- $M,\ s \models\ E\varphi_1 \cup \varphi_2 \Leftrightarrow$ if there exists a path π starting from s such that $\exists s_k \in \pi \mid M,\ s_k \models \varphi_2$ and $\forall s_i \in \pi$ with $i < k$, $M,\ s_i \models \varphi_1$;

- $M,\ s \models\ EG\ \varphi \Leftrightarrow$ if there exists an infinite path π starting from s such that $\forall s_i \in \pi$, $M,\ s_i \models \varphi$.

Notice that the classical boolean connectives ($\vee$, $\Rightarrow$ and $\Leftrightarrow$) and the temporal operators AU, AG, EF, and AF can be derived from the above sets of operators. As an example, let us consider the security and reliability properties presented in Section 4.1, using the set of atomic propositions $AP = \{s, e, f\}$, the meaning of which was explained above. The property "All messages are encrypted before being sent out over the network" can be expressed as $A(\neg sUe)$, meaning that there is no sending until the encryption is performed. The reliability property ("The system eventually recovers from any failure") can be instead expressed as $\neg EFEGf$, meaning that there does not exist a path in which eventually there will be a path in which there is a failure forever.

We formally define Path-CTL by adding a temporal operator to Next-Free CTL that allows the designer to predicate also on finite sequences of events. Path-CTL will be used to describe the constraints that has to be guaranteed by the transparent components to assure the requirements validity. The syntax of the language is formally defined as follows:

$$\phi \rightarrow\ \phi \wedge \phi \mid \neg\phi \mid E\ \phi\ U\ \phi \mid E\ G\ \phi \mid E_p\ G\ \phi \mid p$$

where $p \in AP$, EU and EG are the CTL operators (the above set of derivable operators is still derivable)), E_PG is a fresh temporal operator, that indicates that the arguments, on which it is applied, holds at least in a possible scenario starting from the present until the end of the system behavior, i.e, the final state.

We can define the semantics of Path-CTL on M, a labelled transition system with a unique final state s_F, as defined above. If φ is a formula $M,\ s \models\ \phi$ means that ϕ holds in a state s of the LTS M. Omitting the qCTL operators, we just need to define the semantics of E_pG as follows

$M,\ s \models\ E_pG\ \phi \Leftrightarrow$ if there exists a path π, starting from s and ending in the final state s_F of M, such that, for all s_i in π, $M,\ s_i \models \phi$.

4.2.3 Next-Free CTL model checking of ILTS

Here we present our model-checking algorithm for incomplete models, described as ILTS, against properties expressed Next-Free CTL. Note that we excluded Next operator from CTL in this version of the algorithm. The complete support of CTL is postponed as a future work. The basic idea behind the algorithm is to modify the standard CTL model checking in order to deal with transparent states. The algorithm takes as inputs a Next-Free CTL property and an ILTS. If the ILTS is a regular LTS, it behaves as the traditional approach on regular LTS, while if the ILTS contains transparent states, it computes the set of Path-CTL formulae that shall be guaranteed by the components modeled as transparent states.

More precisely, the algorithm works as follows. First, the Next-Free CTL formula is parsed and its parsing tree is derived. As usual, the leaves of the tree are propositions and the inner nodes are boolean and temporal operators. Similarly to CTL model checking, a bottom-up approach is applied to the tree to calculate the satisfactory states for each sub-formula, starting from the leaves of the tree. For each node of the tree, the set of the states in which the sub-formula holds is calculated by applying Algorithm 1.

The algorithm is invoked for every subtree of the parsing tree. The algorithm takes as inputs a subtree T of the parsing tree (possibly the parsing tree itself), the formula φ, and the ILTS M on which the original formula is evaluated. The tree T is a binary tree, where a node representing a unary operator has a single child, while a node representing a binary operator has two children. We use $T.S$ to refer to the set of states in M that satisfy the formula represented by the current subtree, $T.left$ and $T.right$ to refer to the left and the right subtrees of the current tree (when the root is a binary operator), and $T.son$ to refer to the subtree of the current tree (when the root is a unary operator). The elements of the ILTS M are referred as $M.S$ (states), $M.R$ (regular states), $M.T$ (transparent states), $M.Transitions$ (transition relation), and $M.L$ (labeling function).

The algorithm uses the set X (initialized in line 2) as a local set to store the elements that satisfy φ. Moreover, the set of constraints that are needed to satisfy the formula φ in a transparent state s are saved in a matrix $constr$. Each element $constr(\varphi, s)$ is a set of constraints in the form $[(\psi_1, state_1), \ldots, (\psi_n, state_n)]$, meaning that the formula φ holds in s if the Path-CTL formula ψ_1 holds in $state_1, \ldots$, and the Path-CTL formula ψ_n holds in $state_n$. For example, $constr(EGa, s) = \{[(EGa, s)], [(E_pGa, s), (EGa, s')]\}$ means that the formula EGa holds in the transparent state s either if the formula itself holds in the correspondent component or if the formula E_pGa holds in the correspondent component and EGa holds in the component represented by the transparent state s'. Roughly speaking, the elements of the set are conjunctions and the set is seen as a disjunction of such conjunctions. The evaluation algorithm is based on a switch on the value of

Algorithm 1 Node evaluation

```
evaluate($\varphi, T, M$){
 $X = \varnothing$
 switch ($\varphi$){
  case $\varphi \in AP$ :
   for all $s \in M.S$ { $constr(\varphi, s) = \varnothing$; }
   for all $s \in M.S$ {
      if ($s \in M.R$ && $p \in L(s)$) {
       $X = X \cup \{s\}$;
     }elseif($s \in M.T$){
       $X = X \cup \{s\}$;
       $constr(\varphi, s) = constr(\varphi, s) \cup \{(\Theta p, s)\}$; }}
  case $\varphi = \neg\varphi_1$ :
   for all $s \in M.R - T.son.R${
     $X = X \cup \{s\}$; }
   for all $s \in (T.son.S \cap M.T) \vee (s \in T.son.R \wedge constr(\varphi_1, s) \neq \varnothing)${
     $X = X \cup \{s\}$;
     $constr(\varphi, s) = buildNeg(constr(\varphi_1, s))$; }
 case $\varphi = \varphi_1 \wedge \varphi_2$ :
   for all $s_1 \in T.left.S${
     for all $s_2 \in T.right.S${
       if ($s_1 = s_2$){
        $X = X \cup \{s_1\}$;
        if($constr(\varphi_1, s_1) \neq \varnothing \vee constr(\varphi_2, s_1) \neq \varnothing$){
          $constr(\varphi, s) = ANDCombine(constr(\varphi_1, s_1), constr(\varphi_2, s_1))$; }}}}
  case $\varphi = E\varphi_1 U \varphi_2$ :
   for all $s_2 \in T.right.S${
     $X = X \cup s_2$
     if($s_2 \in T.right.S$){$constr(\varphi, s_2) = resolveRightUntil(\varphi_2, s_2)$}
     $X' = \varnothing$;
     while($X' != X$){
       $X' = X$;
       for all $s_1 \in T.left.S${
         if($\exists s' \in X | (s_1, s') \in M.Transitions$)
           $X = X \cup \{s_1\}$
           $\pi = buildPath(s_1, T.right.S)$
           {$constr(\varphi, s_1) = resolveLeftIUntil(constr(\varphi_1, s_1), \pi)$; }}}}
```

Algorithm 1 The rest of node evaluation algorithm

```
37:   case $\varphi = EG\varphi_1$ :
38:     $S' = \varnothing$;
39:     for all $s \in M.T\{$
40:       $S' = S' \cup \{\{s\}\}; X = X \cup \{s\}$;
41:       $constr(\varphi, s) = resolveOutSCC(constr(\varphi_1, s); \}$
42:     for all $sub_S \in \wp(T.son.S)\{$
43:       if($sub_S$ is a scc)$\{$
44:         $S' = S' \cup \{sub_S\}; X = X \cup sub_S$;
45:         for all $s \in sub_S\{$
46:           $constr(\varphi, s) = resolveInSCC(constr(\varphi_1, s), subS); \}\}\}$
47:     for all $sub \in S' \cup M.T\{$
48:       $X' = sub$
49:         $X'' = \varnothing$;
50:         while($X''! = X'$)$\{$
51:           $X'' = X'$;
52:           for all $s_1 \in T.son.S\{$
53:             if($\exists s' \in X' | (s_1, s') \in M.Transitions$)
54:               $X' = X' \cup \{s_1\}$
55:               $\pi = buildPath(s_1, T.right.S)$
56:               $constr(\varphi, s_1) = resolvePathGlobally(constr(\varphi_1, s_1), \pi); \}\}$
57:       $X = X \cup X'; \}$
58:   $\}$
59: $T.S = X$;
60: $\}$
```

the most external operator in φ (line 3). Considering the grammar of Next-Free CTL, there are five different cases: atomic propositions (lines 4–11), negated formulae (lines 12–17), conjunctions (lines18–24), EU formulae (lines 25–36), and EG formulae (lines 37–58).

If φ is an atomic proposition and T is a leaf, the value of $constr(\varphi, s)$ is initialized for all s. Note that this is the only case in which $constr(\varphi, s)$ is based on the value of the sub-formulae. Then, all the regular states labeled with φ are added to the set of states X in which the formula holds (lines 7-8). Moreover all the transparent states are added to X (line 10), together with an update of the correspondent $constr$ slot. In particular, for each transparent state s, the constraint Θp is added to $constr(\varphi, s)$(line 11). The symbol Θ represents a yet non-identified Path-CTL operator, of which the kind will be resolved in the rest of the algorithm. The operator Θ indicates that a propositional formula, that is apparently evaluated on a state, will be evaluated on a component. If the propositional formula is inside a temporal formula, Θ will be resolved by the semantics of the outer operators.

If T is a subtree of which the root is a $\neg$ operator, i.e., φ is a formula of the form $\neg\varphi_1$, all the regular states that are not in the set of states in which φ_1 holds are added to the set X of states in which φ holds (line 13-14). The transparent states are always added to the set of states in which a formula holds together with a set of constraints (that however could also be unsatisfiable). Thus, every transparent state s is added to X. Moreover, the regular states in which the formula φ_1 conditionally holds are added to X. For both these kinds of states, the correspondent slot $constr(\varphi, s)$ is updated through the function $buildNeg(constr(\varphi_1, s))$ (lines 15-17). This function considers the "negation" of the set of constraints for φ_1 in s. At this stage, $\neg\Theta p$ is changed to $\Theta\neg p$, since the constraint comes from an untimed sub-formula. Note that the set represents a disjunction of constraints, while each element in square bracket represents a conjunction of constraints and this has to be considered in negating the set. For example the negation of $constr(EGa, s)$ considered above is $\{[(\neg EGa, s), (\neg E_pGa, s)], [(\neg EGa, s), (\neg EGa, s')]\}$.

When φ is a formula of the form $\varphi_1 \wedge \varphi_2$ and T is a subtree of which the root is a $\wedge$ operator, all the states that are both in the set of states in which φ_1 and φ_2 hold are added to the set X of states in which φ holds (line 19-23). If an added state contains a constraint w.r.t. the considered sub-formula, the correspondent constraint is built using the function $ANDCombine$ $(constr(\varphi_1, s_1)$, $constr(\varphi_2, s_1))$ (lines 23-24). This function basically considers the "conjunction" of the two sets, by simplifying the elements on the same state in the same constraint. At this stage, the conjunction of the elements Θp and $\Theta p'$ is considered as $\Theta(p\wedge p')$, because both the constraints come from an untimed formula. For example, if $\varphi = EGa \wedge EaUb$, $constr(EGa, s)$ is defined as shown above and $constr(EaUb, s) = \{[(EaUb, s)], [(E_pGa, s), (EaUb, s')]\}$, then $constr(EGa \wedge EaUb, s)$ becomes $\{[(EaUb, s), (EGa, s)], [(E_pGa, s), (EGa, s), (EaUb, s')],$ $[(EaUb, s), (E_pGa, s), (EGa, s')], [(E_pGa, s), (EaUb, s'), (EGa, s')]\}$

If T is a subtree of which the root is an EU operator and φ is a formula of the form $E\varphi_1U\varphi_2$, the procedure is in two steps. First, all the states that are in the set of states in which φ_2 holds (T.right.S) are added to the set X of states in which φ holds. (line 26-27). If an added state s is transparent, the constraint of s for φ is updated using the function $resolveRightUntil(\varphi_2, s)$. This function transforms the elements of the form (x, s) that appears in $constr(\varphi_2, s)$ into $(E\varphi_1Ux, s)$. Note that the algorithm only changes the constraints connected to the current states and not the others on adjacent states of a constrained sequence. At this stage, if x has the form Θp or contains a Θ, the operator Θ is deleted. Second, X is updated by using φ_1 (lines 29-36). More precisely, we update X by adding the states, in which φ_1 holds (condition in line 32) and from which it is possible to reach a state in X (condition in line 33). The idea is that φ_1 holds in such states (these states can be either regular or transparent) and from them it is possible to reach directly one of the states in X. For each added state, the path π that connects it to a state in the set in which φ_2 holds is computed (line 35). The path π is used to enrich the set of constraints that make φ hold in it. For this purpose, the algorithm uses the function $resolveLeftIUntil(constr(\varphi_1, s), \pi)$, which adds to $constr(\varphi, s)$ a constraint composed by the conjunction of all the constraints x that makes φ_1 true in the transparent states of π (except the last one), and updates them in E_pGx. Again, if the original constraints contain Θ, the operator Θ is automatically deleted.

Finally, if T is a subtree of which the root is an EG operator, i.e., φ is a formula of the form $EG\varphi_1$, all the transparent states are added to the set X of the states in which φ holds. Moreover, these states are added as singleton to the set S' that contains all the sets that represent strongly connected components, in which φ_1 always holds. Since, the added states are transparent, the correspondent set of constraints is updated using the function $resolveOutSCC(constr(\varphi_1, s))$ (lines 39-41). This function adds the constraint $EG\varphi_1$ to each of these states. For all the non-elementary possible subset in which φ_1 holds; if the subset is a strongly connected component, the set of the subset is added to S' and the states to X. If there exist transparent states in the added subset, their constraints are updated with the function $resolveInSCC(constr(\varphi_1, s), subS)$ (lines 42-46). This function, for all the states in the subsets, adds a conjunction that includes the constraint E_pGx for each state, where x is the constraint that makes φ_1 hold in that state. Obviously, if the components only contain regular states, this constraint is empty.

As the last step, analogously to what is done for operator EU, X is updated by using φ_1 and S' (lines 47-57). More precisely, starting from each SCC in S', the set of the states in which φ_1 (condition in line 53) holds and from which it is possible to reach a state in which φ holds (condition in line 54) is added to X. Once a transparent node is added, the path π that connects it to the SCC in which φ_1 holds is computed (line 56), and the set of constraints that make φ hold, is updated using $resolvePathGlobally(constr(\varphi_1, s_1), \pi)$. This function works analogously to function $resolveLeftIUntil(constr(\varphi_1, s), \pi)$. In all the functions considered for this case, the operator Θp is automatically deleted.

After the evaluation algorithm is performed on the whole parsing tree, if the satisfactory set of the states for the root contains the initial state of

M, then the property φ holds constrained to $const(\varphi, s_0)$. If there is still an unresolved Θ in this set of constraints, it means that the initial state is a transparent state and also the property φ is untimed. In this case the untimed property that follows Θ has to hold in the initial state of the component, which later would represent the transparent state.

4.2.4 Sketching the correctness of Next-Free CTL model checking

Here we informally describe the correctness of our algorithm by showing the equivalence between the standard checking of CTL and the two-stage checking performed by IMC. Our "proof" technique is based on the semantics of CTL and Path-CTL. Basically, we show that checking a CTL property φ on an ILTS with Algorithm 1 and imposing the obtained Path-CTL formulae to the components that are bound to the transparent states in the ILTS is equivalent to check the same property φ with the classic CTL algorithm on an LTS, obtained by substituting the transparent states in the original ILTS with their correspondent LTS.

Consider an LTS M and an ILTS M', obtained by removing k independent LTSs M_i^T (with $1 \leq i \leq k$) - starting from s_0^i with final state s_F^i - from M and replacing each of them with a transparent state s_i^T. An example, with $k = 2$, is shown in Figure 4.4, where the LTS M_1^T and M_2^T in M are abstracted through s_1^T and s_2^T in M'. A path π of M is called *compatible* with a path π' of M' if and only if π contains exactly the same (and in the same order) regular states of π' and, instead of the transparent states of π', it contains one of the possible paths that cross the graph obtained by substituting the transparent states with the actual components.

Our aim is to show that checking a CTL formula φ on M is equivalent to checking φ on M' using IMC approach.

Let us start by considering formulae of the form $E\varphi_1 U \varphi_2$. Checking the validity of this formula corresponds to check if $M,\ s_0 \models\ E\varphi_1 U \varphi_2$ holds, i.e., if there exists a path π starting from the initial state s_0 such that $\exists s_k \in \pi \mid M,\ s_k \models \varphi_2$ and $\forall s_i \in \pi$ with $i < k$, $M,\ s_j \models \varphi_1$. Regarding the correctness of Algorithm 1, it is enough to show that, given a generic path π' in M', it satisfies $E\varphi_1 U \varphi_2$ and the components corresponding to the transparent states in M' satisfy the constraints obtained by IMC if and only if there exists a path π of M, compatible with π', that satisfies $E\varphi_1 U \varphi_2$.

A generic path π' in M' can be as follows:

1. π' does not contain any transparent state s_i^T;

2. the last state of π' is a transparent state;

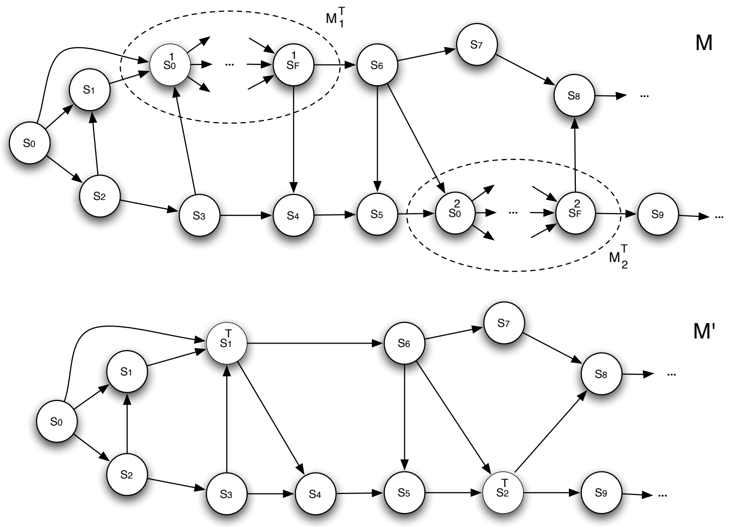


Figure 4.4: *An example of LTS and its corresponding ILTS.*

3. π' contains transparent states, but the last state is not transparent;

4. π' contains transparent states, including the last position.

Obviously, case (4) is a generalization of cases (2) and (3), but since they are more intuitive, we will treat them separately (even if the proof for these cases are included in the proof for case (4)).

The first case is naive. Since there is no transparent state, Algorithm 1 behaves exactly as the classical model checking. The second case corresponds to π' containing only a transparent state at the end. Our algorithm will produce "yes" only if for all s_x in π' (excluded the last $s_{|\pi'|}$) $M', s_x \models \varphi_1$, exactly as required by the classical model checking algorithm. Moreover our algorithm will impose that $E\varphi_1 U\varphi_2$ holds in the component corresponding to $s_{|\pi'|}$ and this will happen only if exists a path π in M compatible with π' that satisfies $E\varphi_1 \cup \varphi_2$. The third case considers a path π' that contains a number of transparent states, but not at the end. Our algorithm will produce "yes" only if for all non-transient state s_x in π' (excluded the last $s_{|\pi'|}$) $M', s_x \models \varphi_1$, and $M', s_{|\pi'|} \models \varphi_2$. Moreover our algorithm will impose that $E_pG\varphi_1$ holds in the component corresponding to the transparent state of π'. All these requirements are satisfied if there exists a path π in M compatible with π' that satisfies $E\varphi_1 U\varphi_2$.
The last case is the most general case and corresponds to π' containing a

number of transparent states, including the end. Our algorithm on such a path would first label the state with φ, using only φ_2. Among all the possible constraints that the labeling imposes, for the proof, we are only interested to the sets that include all the states through the end of π'[1]. So, if $\pi' = s_0, s_1, ..., s_n$ and the sequence of transparent states in it is $[s'_1, \ldots, s'_m]$, for all $0 \leq i \leq n-1$, the set $constr(\varphi_2, s_i)$ can contain the constraint $[(sub_{\varphi_2}, s'_j)$, $(sub_{\varphi_2}, s'_{j+1})$, $\ldots$, $(sub_{\varphi_2}, s'_{m-1})$, $(\varphi_2, s'_m)]$, where s'_j is the first transparent state after s_i in π' and sub_{φ_2} is a subcondition needed to make φ_2 true in the current state. Moreover, $constr(\varphi_2, s_n)$ contains the constraint $[(\varphi_2, s_n)]$, where s_n is exactly the last transparent state s'_m. When our algorithm starts the labeling using also φ_1, each of the above constraints can be used to compute $constr(E\varphi_1 U\varphi_2, s_0)$, adding constraints of the form $[(E_pG\varphi_1, s'_1)$, $\ldots$, $(E_pG\varphi_1, s'_{j-1})$, $(E(\varphi_1 U sub_{\varphi_2}), s'_j)$, $(sub_{\varphi_2}, s'_{j+1})$, $\ldots$, $(sub_{\varphi_2}, s'_{m-1})$, $(\varphi_2, s'_m)]$. Moreover, if such a constraint exists, the algorithm checks that all the regular states before the j-th transparent state satisfy φ_1 and all the regular states after the j-th transparent state satisfy sub_{φ_2}. A compatible path π satisfies $E\varphi_1 \cup \varphi_2$ if and only if it satisfies one of the previous constraints.

An analogous reasoning can be applied to $EG\ \varphi$, while the atomic proposition case and boolean connectors need to be treated differently. When $\varphi \in AP$, φ holds in M if s_0 is labeled with φ. If s_0 is a regular state, then our algorithm will check exactly the same. If instead s_0 is included in an LTS substituted with a transparent state, the algorithm will come up with the constraint, that is exactly the same condition checked by the classical algorithm. Moreover, our algorithm deals with the boolean connectors as the classical one, only modifying the previously obtained constraints according to the connector semantics.

4.2.5 Path-CTL model checking

To verify a Path-CTL property on an LTS with unique final state, we need to observe that LTS with a unique final state is a particular case of LTS, while the final state does not influence the verification of classical CTL properties. Since Path-CTL is Next-Free CTL with an extra temporal operator E_pG, we only need to extend the classical CTL algorithm to deal with this new operator. Algorithm 2 shows a fragment of an evaluation function to deal with formulae φ of the form $E_pG\varphi_1$. The fragment uses the same notation and structure of Algorithm 1. The idea is that, starting from an LTS with final state M, the algorithm builds M' by deleting the states

[1]We are looking at the satisfiability using the whole path; all the subpaths are considered separately as one of the possible four mentioned scenarios.

where φ_1 does not hold (line 3). Then, a state s of M' is added to the set of states in which φ holds (line 7) if the final state s_F belongs to M' (line 4) and there exists at least a path from s to s_F in M' (line 6). This check can be done easily with a breadth-first search in $O(|M'.S|)$, making the overall evaluation $O(|M'.S|^2)$.

Algorithm 2 Checking formulae of the form $E_pG\varphi$

```
1: case($\varphi = E_pG\varphi_1$) :
2:   $S' = \{s \in M.S | s \in T.son.S\}$;
3:   $M' = M|_{S \leftarrow S'}$;
4:   if $s_F \in S'\{$
5:     for all $s \in S'\{$
6:       if $SearchPaths(s, s_F, M')\{$
7:         $X = X \cup \{s\}; \}\}\}$
```

4.3 Experimental Results

In this section, we present the applicability and scalability of the proposed approach in practice.

4.3.1 Tool Support and Applicability

We have developed a prototype tool to verify ILTSs against properties expressed in CTL according to the algorithm presented earlier. The inputs of the tool are two files, which contain an ILTS and the CTL property. The tool is capable to verify the property and report the output as a set of solutions[2]. Solutions consist of Path-CTL properties that constrain the transparent states. The tool is also able to verify LTSs against constraints generated as Path-CTL.

To demonstrate the applicability of our approach, we used the tool to verify the ILTS of the example against two requirements. Regarding the security property $A(\neg s \cup e)$, the model checker returns two solutions that constrain a possible specification of the transparent state (state 5) to satisfy at least one of the solutions. The first solution is $\{S_5 \models A(\neg s \cup e)\}$, which means that the same property shall hold also in S_5. The second solution is $\{S_5 \models A_pG\neg s\}$. This property enforces the paths between the start and end states of S_5 specification to be labeled with $\neg s$, which prohibits Encryptor component to send any message over the network. Applying

[2]The tool is available online: `https://sites.google.com/site/amirsharifloo/tool-lover`

the verification algorithm to the second property returns only one solution: $\{S_5 \models \neg EFEGf\}$. Therefore, any component that is bound at run time to play the role of Encryptor shall satisfy this Path-CTL property.

4.3.2 Scalability

To see how our approach scales up with respect to the number of regular and transparent states, we performed a scalability experiment. To do so, we generated different models by concatenating the running example. Concatenation here means to produce a new ILTS by simply connecting the last state (state 15) of the ILTS to the first state of another copy of the ILTS. For example, the first concatenation results in an ILTS with 30 states in which two states are transparent. This way we generated larger models and applied the tool to verify the properties.

Figure 4.5 illustrates the result, which is obtained by running the experiment 100 times and computing the average. The result shows that the verification time of both properties exponentially grow. However, the verification time of the nested property grows faster as the number of states increases. The machine we used for the experiments had the following characteristics: OS = Mac, CPU=2.4 GHz Core 2 Duo, and RAM=4 GB.

Although in general the verification cost of the algorithm exponentially grows with respect to the number of transparent states, the specification topology is a key parameter that can significantly affect the total amount of the computation. Obviously, it is more than a simple verification performed by an LTS model checker, since the algorithm calculates constraints, considering the combinations. Moreover, our tool is a prototype and the result can be improved by applying further optimizations. Moreover, the constraints can be checked in parallel in order to speed up the verification.

4.4 Verification of open adaptive systems at run time

We show that IMC can be interestingly applied to formally check the properties of *open* adaptive systems, in which parts of specification may dynamically change at run time. We propose *LOVER* [3] as a two-phase framework, in which the verification process is divided into phases: design time and run time. At design time, the changing components of the system are identified and abstracted away from the specification by replacing them with transparent states. The new specification, that represents the invariant behavior of the system, is verified against the system properties. As the result, a set of

[3]LOVER stands for Light-weight fOrmal Verification of self-adaptivE systems at Run time

Transparent	State	$A(\neg s \cup e)$	$\neg E\Diamond E\Box f$
1	15	0.105637	0.079811
4	60	0.76972	0.702177
7	105	3.156306	5.841801
10	150	8.659444	24.611509
13	195	19.197839	70.304578
16	240	36.051059	161.264
19	285	59.829017	326.778

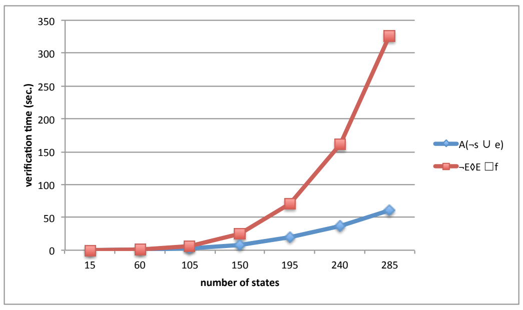


Figure 4.5: *The verification time for the properties (The table provides the precise values shown in the diagram.)*

constraints for changing components are produced that guarantee the satisfaction of the properties. At run time, the dynamic components are verified against the constraints in order to check the satisfaction of the properties.

Differently from the classic model checking, LOVER deals with incomplete models, where a set of components are unspecified at design time and are known only at run time. Obviously, the classical techniques could be applied by checking the system every time the bindings (unspecified at design time) are resolved or changed. Indeed, the time and space required for the verification could be considerable, and since some bindings are resolved only while the system is operating, the total overhead in resolving them should be kept as small as possible.

To overcome these limitations, we propose LOVER that allows the designer to verify the incomplete system specification at design time and generates a set of constraints for the unspecified components. Those constraints are verified at run time whenever the component specifications become available. An overall view of LOVER is given in Figure 4.6. At design time, the incomplete system is described as a particular kind of LTS, where some states are transparent w.r.t. the labels. This model is then checked against a desired Next-Free CTL property. The result of the verification could be "yes", "no" or "conditionally yes". The last option gives the set of constraints that has to be satisfied by the unspecified components such that the whole system satisfies the given property. These constraints are expressed in Path-CTL, an extension of Next-Free CTL that allows the specification of properties also over finite paths. The constraints are verified by a Path-CTL model checker, which can be obtained by a simple extension to any CTL model checker, such as NuSMV [15].

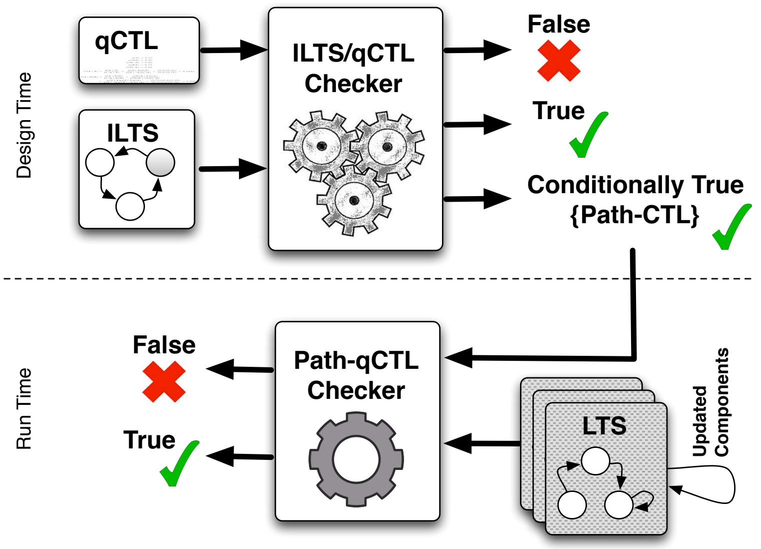


Figure 4.6: *LOVER Framework*

4.5 Related Work

Modeling evolvable specifications is studied by Sampath et al. [78], in which a new formalism called Structured Transition Systems is presented to ease the evolutionary requirements modeling and refinement. On the contrary, we exploit the application of an existing formalism like Statecharts as a widely known formalism that comes with nice features supporting modularity and incremental refinement. Shaker et al. [80] propose a feature-oriented approach to specifying the requirements of Software Product Lines (SPL) in order to facilitate the process of adding new features to existing specifications. This way the specification of SPL is more flexible to possible changes. However, the approach still lacks the support for any analysis.

Analyzing models with unknown elements during development lifecycle has been studied in the past years. Salay et al. [75] addresses the problem of expressing uncertainty in requirements specifications. In particular, they study this problem for early stage specification e.g. i* models [90] and class diagrams, and propose an approach grounded upon a previous work on partial modeling [31, 76] to reason on the models. They describe how to construct partial models from possible design alternatives. The main idea is to annotate the elements of models, that exist in only some of alternative, with Maybe tags and then to apply SAT-based analysis techniques to check First-Order Logic properties. The analysis output produces three possible values: True, False, and Maybe.

Uchitel et al. [89] discuss how scenarios can be viewed as partial models of system behaviors that can be incrementally added during the development iterations. Moreover, they describe how to detect enabled, prescribed, and unknown system behaviors by merging various system scenarios. Enabled behaviors are the behaviors that system is supposed to perform, while the prescribed behaviors are prohibited. The unknown behaviors do not violate any conditions but are not declared and cannot be inferred from the specification. To specify such behaviors, PLTL (*Partial LTL*) is used to describe the system specification. In [88], the authors take a further step by synthesizing scenarios and safety properties to construct behavioral models in the form of *Modal Transition Systems*.

Bianculli et al. [8] present an approach to deriving forbidden behaviors of external services given a requirements specification. More precisely, they use a behavioral model, that describes the set of traces that leads to error state, and given a set of external services and their fine-grain operations, they provide a heuristic technique to produce the behaviors, in terms of LTS, that external services shall avoid.

Adler et al. [1] propose an approach to modularly design and model adaptive embedded systems such that the system specification is suitable for verification analysis. The approach distinguishes between the part of the system that supports the functionality and the part that manages the adaptation, and focuses on specifying the adaptation behavior in order to verify the stability property of the adaptation process. Theorem proving techniques e.g. Isabelle/HOL are employed to verify the properties. The approach is extended in [79] to verify system properties with respect to environment constraints. To this end, the interaction among the system and the environment is modeled and is verified that the system properties are guaranteed assuming a maximal environment. This approach assumes that all the environmental behaviors can be predetermined in advanced so the verification of the properties are performed at design time. Although applying modular techniques reduces the verification costs, the approach assumes that the whole knowledge on the specification and the adaptations is available at design time.

Păsăreanu et al. [22,34] propose an approach to automatically generating assumptions for the environments of a component, and apply the technique for compositional verification. The output of the approach describes the environments in which a component will satisfy the expected properties. Our approach is different in the point that there exists a couple of unspecified components that make the specification incomplete and the verification unfeasible. What we do is to enforce those components with some constraints such that the global properties hold.

4.6 Conclusion

Incomplete Model Checking allows us to reason about partial specifications, and extends classic model checking by synthesizing constrains such that the global properties hold. The application of such model checking technique goes beyond design-time verification, and can effectively apply to check adaptive systems with dynamic components. It is important to note that IMC works the same as classic model checking in case the specification is complete.

CHAPTER 5

AGAVE: A Methodology for Incremental Verification

"Most People like to believe something is or not true. Great scientists tolerate ambiguity very well. They believe the theory enough to go ahead; they doubt it enough to notice the errors and faults so they can step forward and create the new replacement theory." Richard Hamming

In this chapter, we extend IMC paradigm presented earlier to provide an incremental verification environment AGAVE, that enables developers to use model checking as they refine system models. Unlike ILTS specifications, AGAVE is designated to handle also high-level models, which more often appear in iterative and incremental software development. The methodology deals with incomplete specifications, where some parts are left unspecified and may later be further elaborated at development time, or even may be left as components dynamically deployable at run time. Following this approach, the verification not only is able to check whether or not a specification can satisfy a given property but also generates a set of sub-properties for the missing components to ensure satisfaction of the global property. In the next iterations and when the developer is ready to elaborate the missing components, one only needs to care about the sub-properties of the components and not the system's global properties. This is a great advantage that simplifies modeling, encourages exploration and verification of alternative designs, supports decision making based on rigorous analysis, and drastically reduces the verification cost. AGAVE can be very useful to define the expected properties that must be exhibited by off-the-shelf components (*COTS*) to be hired from a third party, following a *design by contract* approach [65].

AGAVE consists of an incremental process model through which the verification is decomposed and performed in iterative steps. To realize this approach, we explain how the Statecharts language can be equipped with incremental verification by providing a model-checking algorithm. Statecharts are chosen because of their expressive power as well as their hierarchical decomposition, which naturally support incremental refinement.

5.1 Overview

Incremental development consists of a series of developments of partial (incomplete) system models, where additional details are progressively added. A typical way is to progress through subsequent *refinement steps*, which allow engineers to develop a complete and detailed system model starting from a high-level abstract model and progressing through refinements where a more detailed structure is given, down to the desired level of detail.

AGAVE is intended to support the iterative and incremental development of system models, by providing an analysis method that can be applied incrementally while the model is built. The overall idea is to use a state-based modeling formalism to specify the system and to proceed with incremental steps. Initially, system is represented as a model M, in which all its most significant components and their interactions are represented. Together with the system, the set Φ of properties the system under analysis should meet are specified. At this stage, the components that need to be further detailed are represented as simple states, marked with a T to indicate that they will be specified later with a state-based model as well[1]. Then model M is verified against each property in Φ. This form of delayed and separate specification can be used to support both progressive knowledge acquisition and also to manage the separate exploration of possible design alternatives.

[1]Since a state can represent both a system state and a component, in the paper we will use states and components as synonyms.

If a system is fully specified, verification either succeeds or it fails. However if it contains unspecified components, it generate a set of properties that they must satisfy to get the property holds in the whole system. This way, once these components are specified, the verification algorithm does not check again the whole specification, but just their new specification against the properties previously generated. This procedure is recursively repeated every time an unspecified state is refined.

The overall approach is exemplified in Figure 5.1. The *level 1* of model M is verified against a high-level requirement formalized by a property φ. M includes two unspecified components C_1 and C_2. As a result of verification, the algorithm produces a set of properties that have to be satisfied by C_1 and C_2. In the second step, once C_1 and C_2 are specified, their models are checked against the properties derived in the previous step. Notice that, while in the figure it seems that a single property is derived for each component, the set may contain more than one property for each state. At *level 2*, the model that refines C_2 does not contain any unspecified components, and hence it is verified by classic model-checking techniques, while the model that refines C_1 is analyzed by applying the AGAVE approach, which computes the properties that must be satisfied by the model that eventually will refine the unspecified state C_{11}.

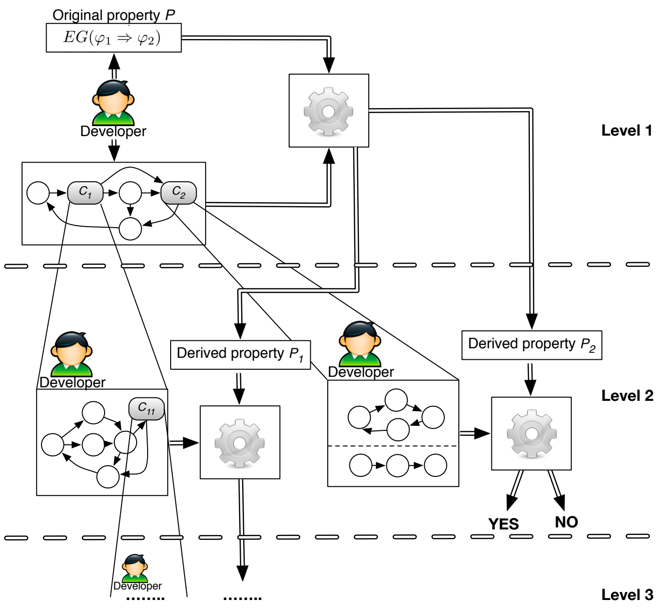


Figure 5.1: *An overview of the* AGAVE *methodology.*

AGAVE provides advantages both from the modeling and the verification viewpoints. Indeed, from the modeling aspect, working at different abstraction levels helps progressively delving into details that need to be explored. Concerning verification, AGAVE supports dealing with incremental verification. Alternative refinements can be separately verified without flattening the entire specification and completely re-analyzing it, even if a change has only been made in refinement step.

The AGAVE approach is conceptually independent of the details of the specific state-machine based modeling formalisms and property languages. To be practically usable and effective, however, AGAVE is further developed for Statecharts and Path-CTL property language.

Statecharts [43] are a structured graphical formalism used to describe reactive systems, such as communication protocols, digital control units, and software systems. It is a state-based formalism that is well-suited for incremental modeling, due to its hierarchical structure.

The idea is that the input Statechart model is preprocessed and all the composite states, i.e. the state that represent themselves a Statechart, are seen as transparent, ignoring that they are already present in the model. AGAVE is applied on the transformed model and this process is recursively re-iterated on the Statecharts using the properties derived by AGAVE in the previous steps. The iterative process terminates when the verification is performed on all the composite states.

5.2 Statecharts

Statecharts are an extension of finite state machines, enriched with hierarchy and concurrency. The former is used to model the system at different levels of abstraction by refining states through a sub-Statechart or the composition of sub-Statecharts. The latter supports the definition of two or more Statecharts running in parallel, synchronized through the use of global controlled variables. Given their structured and parallel nature, Statecharts are a suitable tool for modeling complex systems in a compact way.

Since the original work by Harel [43], which introduced Statecharts informally, the increasing popularity of the formalism over the years prompted different definitions of the semantics and proposals for extensions [44, 54, 61]. Hereafter, we consider the basic elements of the original definition of Statecharts which include its most popular features, ignoring time actions, history, special events (e.g., events generated when a state is entered or exited) and special actions (e.g., start action, history clear, deep clear) and we refer to the STATEMATE semantics, proposed by Harel [44].

Before formally introducing Statecharts, some preliminary definitions are needed. Given a set of atomic proposition AP , a *condition* c and an *action* a, over AP, are defined, respectively, as

$$c \rightarrow p \mid \neg c \mid c \wedge c,$$

$$a \rightarrow p = false \mid p = true \mid neg(i),$$

where $p \in AP$ and neg is an operator that negates the truth value of p. $C(AP)$ and $A(AP)$ are the set of conditions and actions over AP, respectively.

AP is, in general, partitioned in two subsets: E, representing events, and I, representing the internal controlled propositions of the system i.e., $AP = E \cup I$ and $E \cap I = \varnothing$. A Statechart over AP is defined as a tuple $S = \langle Q, q_0, q_F, St, \rho, \tau \rangle$, where

- Q is a finite set of states that can be themselves Statecharts, also called *chart-states* [92];
- $q_0 \in Q$ is the initial state;
- $q_F \in Q$ is the final state;
- St is a finite set of Statecharts;
- $\rho \subseteq (Q - \{q_0, q_F\}) \times \{AND, OR\} \times \wp(St)$ is the hierarchical relation, used to decompose states into sub-states ($\wp(St)$ denotes the power set of St). More precisely, each state of S can be decomposed through an and- or an or-decomposition of one or more Statecharts;
- $\tau : (Q - \{q_F\}) \times E \times C(I) \rightarrow (Q - \{q_0\}) \times A(I)$ is the transition relation that associates a triple composed of a state in Q, an event $e \in E$, and a condition c over I with a pair composed of a state in Q and an action over I. Each transition is represented by a directed arc between states and is labeled with an ECA rule (event, condition, and action—these will be defined below).

Given a state $q \in Q$ and a ρ-related Statechart $s \in St$, q is called *parent* of s (and of the states in s), and s is the *child* of q. The ρ-related Statechart is also called a sub-Statechart. States that are not ρ-related with any Statecharts are called *basic* states, and all the others are called *composite*. As defined in the ρ relation, composite states can be refined into *and-* or *or*-decomposition of Statecharts. The root states are those states that belong to a Statechart with no parents; they can be simple or composite. By definition, initial and final states are basic and they do not have incoming and outgoing transitions, respectively. The former is often represented as a simple large dot and the latter as an encircled dot.

A Statechart transition is labeled with an ECA rule, defined as a triple: event e, condition c, action a. The event e is an element of the set $E \subseteq AP$, and represents an environmental proposition that has to be true to enable the transition to fire. Since elements in E are events, they are true only when explicitly indicated, and their truth value cannot be controlled. The condition c and the action a are elements of $C(I)$ and $A(I)$, respectively, and are built as defined above. Informally, c is a boolean expression on propositions in $I \in AP$, which as to be true for the transition to be enabled, and a represents the set of changes actuated if the transition is performed. As a shortcut in the graphical notation, either the event or the condition, as well as the action can be missing.

Basically, Statecharts are finite state automata, in which the execution starts from the initial state by firing a transition. Transitions are triggered by an event and are guarded by a condition on the internal propositions. Once a transition to a new state is executed, the action indicated on the fired transition is performed, modifying the values of some internal proposition.

The composite states represent the components that are hierarchically specified through decomposition. If the current state s is or-decomposed, being in s means being in one of the states of the refining sub-Statechart. In addition, outgoing transitions from s become active only when the execution of the sub-Statechart is over; that is, it has reached its final state. If the state s is and-decomposed state, s is refined by a set of two or more sub-Statecharts, that become active when s is the current state. Similarly to the case of or-decomposition, outgoing transitions from s become active only when all the sub-Statecharts in the decomposition have reached their final state.

More formally, the STATEMATE semantics describes the behavior of a system as a sequence of configurations. A configuration c describes a snapshot of the system (i.e., the current state(s), the values of the internal variables, and the active events). Since the current state may be a composite state, the configuration should also include the configuration of the executing sub-Statecharts. Formally, the set c_s of active states must satisfy the following rules:

- c_s must contain the root state of the Statechart that contains the current basic state(s);

- If c_s contains an or-decomposition, it must also contain exactly one state of the sub-Statechart [2];

- If c_s contains an and-decomposition, it must also contain one state for each sub-Statechart;

The initial configuration c_0 contains the initial state q_0 as active state and the initial values of the internal propositions are all set to false, if they are not explicitly initialized.

If the system configuration is c, the system may evolve by firing a transition. A transition is triggered by an external event and by the satisfaction of its condition. All the transitions outgoing the set of basic current states in the current configuration are executable once triggered. As for the transitions outgoing a composite current state, we need to distinguish between the case of an or-decomposition or an and-decomposition. In the former case, the outgoing transition from the composite state can be executed only if the sub-Statechart is in the final state. In the latter case, it can be executed only if *all* sub-Statecharts are in a final state.

Moreover, the STATEMATE semantics regulates the sequence of configurations following these assumptions:

- The changes that occur during a transition can be sensed only after its completion. This means that if an action a is performed during the transition between configuration n and configuration $n + 1$, the result of the action is visible in configuration $n + 1$.

- The occurrence of an event is visible only for the duration of the transition in which it happens. If an event e occurs in configuration n and causes a transition to configuration $n+1$, the event is no longer visible in configuration $n + 1$.

- In each configuration a maximal subset of non-conflicting transitions is always executed, i.e., given the current configuration, the maximal number of enabled transitions is fired.

5.3 Verification

This section describes how formally specified requirements can be verified by AGAVE for system models described through Statecharts. We discuss verification in the context of an iterative and agile development style, where some states can be temporarily left unspecified and where the designer can

[2] According to STATEMATE, or-decomposition has an $exclusive$ semantics.

quickly explore alternative decompositions to evaluate their possible impact on requirements satisfaction. We assume that the Statechart may contain basic states, composite states, and unspecified states, that are states whose internal structure is currently unknown and will be specified later. Composite and unknown states are called *transparent*. If a transparent state is refined by a sub-Statechart, this is allowed to have transparent states to support partial specification at all levels.

Together with the Statechart specification M, we provide a set of properties Φ expressed in Path-CTL, which describe the requirements the specification is expected to satisfy. M is verified against every $\varphi \in \Phi$ using the Algorithm 1. The verification, however, may not yield a definite result (TRUE or FALSE), since the result may depend on the yet unknown behavior of transparent states. In this case, the algorithm calculates the set of constraints for the future refinement of the transparent states to guarantee the satisfaction of the initial property. The algorithm behaves in the same way for unknown and composite states, but in the latter case, the constraints produced by the algorithm, can be immediately checked on the sub-Statechart. Hence, this technique allows the developer not only to verify partially specified systems, but also to deal efficiently with the verification of completely defined Statecharts, by splitting the verification in multiple levels.

This approach performs the exploration of possible different refinements efficiently in an incremental manner that only analyzes the alternative refinements.

Algorithm 1 Statechart Verification

```
1:  function CHECK($M$, $\varphi$)
2:      $ilts$ = transformSC2Ilts($M$)
3:      $result$ = verify($ilts$, $\varphi$)
4:      if $result.isUnconditional() = T$ then
5:          return $result$;
6:      end if
7:      for ($trans\_state$,$sub\_p$) in extract($result.cons$) do
8:          if $trans\_state$ is composite then
9:              $subSC$ = load_subSc($trans\_state$, $M$)
10:             $sub\_result$ = CHECK($subSC$, $sub\_p$)
11:             if $sub\_result.isUnconditional() = F$ then
12:                 $result$ = update($result$, $sub\_result$)
13:             end if
14:         end if
15:     end for
16:     return $result$
17: end function
```

Algorithm 1 works through a number of steps. First (line 2), the model M, which represents a particular level of the Statechart, is translated into the equivalent labeled transition system (ILTS). This ILTS is verified against the property φ and the verification outcome is returned as $result$. If no constraint is generated ($result.isUnconditional$ equals $true$), the algorithm exits.

Otherwise, $result.cons$ contains the set of constraints that shall be satisfied by the unknown components to make the property hold, and the rest of the algorithm iteratively extracts and analyzes each of these constraints (line 7–15). The constraint (sub_p) of each transparent state ($trans\_state$) is checked by recursively invoking the same algorithm, feeding the generated constraint and the corresponding Statechart $subSC$. The verification result is updated and gradually completed with the outcomes of these inner verifications.

5.3.1 Statecharts-to-ILTS Transformation

Hereafter, we discuss how to transform Statecharts (with transparent states) into an equivalent ILTS representation. To do that, we first need to apply two preprocessing steps. The first step eliminates transparent states by mapping each of them onto two basic states connected by an unlabeled transition, similarly called *transparent* transition. A transparent transition represents the internal behavior of the corresponding transparent state. The set of incoming transitions that reach the original transparent state are connected to the source state of the transparent transition, while the outgoing transitions depart from the destination state, as shown in Figure 5.2.

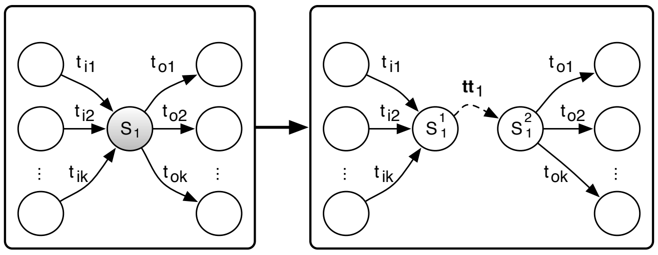


Figure 5.2: *Replacing transparent states with transparent transitions*

The second preprocessing regards and states. and states refine a state into two or more sub-Statecharts that are executed in parallel. Our algorithm replaces these sub-Statecharts with a single Statechart whose set of states is the Cartesian product of the sets of states of the sub-Statecharts, and transitions represent all possible interleavings. Transparent transitions in the source Statecharts remain transparent also in the generated target Statechart.

At this point, we can generate an equivalent ILTS through two basic steps: *producing the graph* and *labeling the ILTS states*.

To produce the ILTS graph structure, each transition of the Statechart is transformed into an ILTS state. If the transition is transparent, the generated state is also transparent and labeled with T. Since ILTS states represent transitions of the original Statecharts, two states are connected in the ILTS only if the corresponding transitions can be executed sequentially, one after the other, in the original Statechart. The algorithm also creates two additional ILTS states: the initial and the final one, respectively connected to all the ILTS states that represent transitions of the original Statechart connected to the initial and final state of the Statechart. Figure 5.3 shows a Statechart fragment and the corresponding target ILTS structure (drawn with dashed lines)[3].

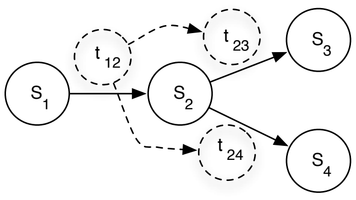


Figure 5.3: *Transforming transitions into states*

Once the structure is created, we need to perform the (possibly incomplete) labeling of ILTS states. We recall that the labels of ILTS states describe the set of propositions that are true in the states, if they are known. The labeling is performed according to the following procedure. First, each state in the target ILTS is labeled with the events that trigger the corresponding transition in the source Statechart. Concerning the actions associated with a Statechart transition, since they can modify the values of a set of atomic propositions, the ones that are true are used to label the corresponding ILTS state (let us call it s).

At this stage, we need to perform further labeling of s, since we only took into account the actions, which tell us which atomic propositions changed by performing the corresponding Statechart transition and this may obviously not include all the propositions true in s. To complete the label-

[3]Note that in the example, the initial and the final states are not created since they are not present in the original Statechart.

ing, we also need to add the propositions in the previous states (the ones whose outgoing transitions lead to s) that must be propagated to s because they did not change during the transition. Notice that, in general, since a state may be reached with different paths, after the labeling, it may contain contradictory propositions (e.g., one may contain p and another $\neg p$). In this case, we need to replace the state with the duplicate states s_1^1 and s_1^2, one including label with p and the other including $\neg p$. The successor states of s are then connected to s_1^1 and s_1^2 and possible further propagation with state duplication may then occur.

Finally, we need to consider the case of an ILTS transparent state st. For simplicity, let us first assume that the possible refinements of transparent states in the source Statechart do not modify the truth value of propositions. This assumption does not mean that the transitions of the sub-Statechart cannot modify such values, but just requires that after its completion the propositions are set back to their initial values.

Let us further assume that there is only one ILTS state (s_{i1}), whose outgoing transition leads to the transparent ILTS state st. Then, all the basic ILTS states $s_{o2}, s_{o3}, \dots, s_{on}$ that directly follow the transparent state st are labeled with the set of atomic propositions that are true in s_{i1}, with the obvious exception of the propositions modified by the Statechart transitions associated with the ILTS states $s_{o2}, s_{o3}, \dots, s_{on}$. If we now consider the case when more than one ILTS state (s_{i1}, s_{i2},..., s_{in}) preceeds st, as for the non transparent ILTS state, contradictory situations have to be considered. If for example, s_{i1}, s_{i2},...,s_{in} contain contradictory propositions, we need to duplicate st and handle this case similarly to the propagation case described earlier. It is also possible to relax the first assumption, considering the case in which the transparent transition can modify the value of some atomic proposition. To sketch the approach consider the case, where a transparent ILTS state st, associated with the transparent transition tt of the original Statechart and connected to the state s_{o2}, can modify the value of the atomic proposition p. Then s_{o2} is split in two states: s_{o2}^1 where p is true, and s_{o2}^2 where p is false. This splitting is motivated by the need for considering all the possible value of p after the execution of the component in st, since this value is not a priori known.

After all states have been fully labeled, we need one final state to take into account the effect of the condition of the original Statechart's transitions. For each transition t of the original Statechart, represented as a state in ILTS (s), we check the labeling of the previous ILTS states (say s_{i1}). If this labeling is consistent with the condition of the transition t, then the connection between s_{i1} and s is kept. If this is not the case, the connection is removed. If all the incoming connections to a state are removed, the state itself is also removed.

5.3.2 ILTS/Path-CTL Verification

The model-checking algorithm verifies the Path-CTL properties against the ILTS previously generated. The Path-CTL properties could be the one derived by a previous step of the verification or the original stated in Φ. Notice that, even if the properties in Φ were stated on the Statechart and not on the ILTS, they do not need to be changed, since the paths on the ILTS are equivalent to the possible sequences of configuration in the original Statechart.

Because of the and-decomposition of Statecharts, that are resolved by performing the Cartesian product, and the duplication caused by the splitting in the labeling procedure, a transparent state s in the original Statechart may be represented by different transparent states in the ILTS. In order to check the derived property for the transparent component on the correct element, the algorithm must aggregate the constraints of these ILTS transparent states, generating a single constraint for the original Statechart state s. Basically the aggregation consists in a disjunction of the generated constraints, simplified by removing the duplicated constraints.

5.4 Railway Crossing System

In this section, we describe the application of our approach through an extension of the classic Railway Crossing System (RCS) [73]. The main goal of RCS is to control trains and gates, such that a train never crosses a gate when it is open (a high-level is shown in Fig. 5.4). This requirement is a safety property, whose violation may lead to accidents.

There are three sensors (A, B, and C) placed on the track to detect when a train approaches, crosses and leaves the gate. Another property that shall be satisfied is that the train can cross the gate only if it obtains permission from a central authority ("central station"). The central station manages the railway lines, and it has its own policies to regulate the dispatching of permissions. For example, if an emergency situation is detected and the train is approaching, the central station will not give to the train the permission to cross, and the train has to stop.

The high-level modeling of this system leads to the first-level Statechart in Figure 5.5, which consists of two concurrent components: gate and train, which interact together via transponders. The gate may be in one of two states s_4 and s_5. The gate can switch between these two states by acting on the variable *open*, and according to the modes of the train. If the train is

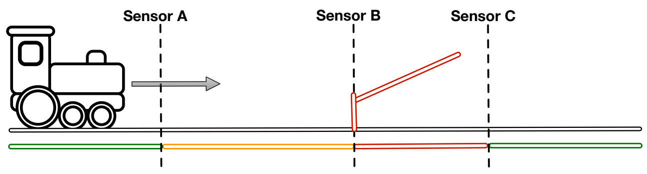


Figure 5.4: *Railway Crossing System*

switched to *approaching* mode, transition t_4 is activated and the gate can be closed. When the train returns to *traveling* mode, transition t_5 is activated and the gate can be opened.

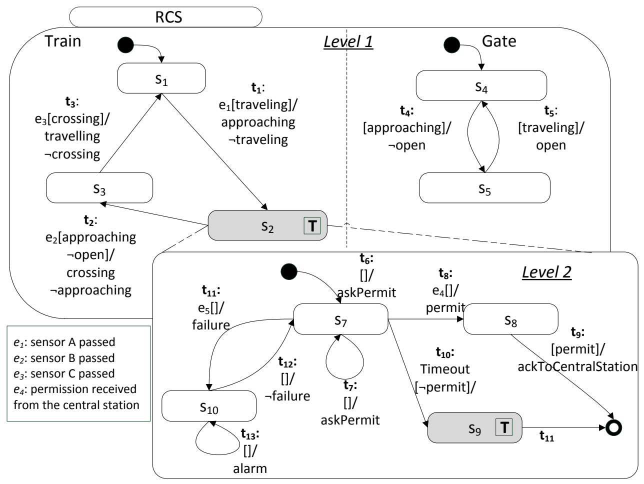


Figure 5.5: *The Statecharts of RCS*

The train modes are represented by three boolean variables: *traveling*, *approaching* and *crossing*, which change as the three sensors are passed. At the beginning, the train is in *traveling* mode (state s_1). When the train passes sensor A, event e_1 is generated, and the train moves to *approaching* mode (state s_2). Analogously, when sensor B is passed, it generates event e_2 and the train switches to *crossing* (state s_3). This transition is performed only when the gate is closed. Finally, when event e_3 is generated, the train has completely crossed the gate, and the mode is changed back to *traveling*. State s_2 is considered as a transparent state, since its refinement is postponed to next modeling phase. In fact, when the train starts *approaching*, different operations can be executed and different component can be activated.

In our case, the train, once approaching, must communicate with the central station to receive the permission before crossing the gate. This requirement can be expressed by two Path-CTL $\varphi_a = AF(crossing)$ and $\varphi'_a = \neg E(\neg permit\ U\ crossing)$, where AF stands for *all path eventually* and EU stands for *exists a path until*. The former is a liveness property stating that in any case the train will cross the gate, while the latter is a safety property stating that there is no behavior in which the train crosses without receiving the permission. Furthermore, we consider another reliability requirement which guarantees that the system recovers from any failure. This property can be expressed in Path-CTL as $\varphi_b = \neg\ EF(EG\ failure)$. Due to lack of space, we only focus on φ'_a and φ_b in the rest of the paper.

To reduce development risks and anticipate possible requirements violations in an early development stage, we would like to check if the high-level specification (though incomplete) satisfies these requirements. The algorithm described in the previous section transforms the first-level Statechart shown in Fig. 5.5, into the ILTS illustrated in Fig. 5.6. The ILTS is checked against the property φ'_a and φ_b. The first property leads to the following constraints for state s_2: $\varphi'_{a1} = \neg E(\neg permit\ U\ crossing)$ and $\varphi'_{a2} = \neg(EpG(\neg permit))$. φ'_a holds in the Statechart only if both of these constraints are satisfied by a further refinement of s_2. The verification of the second property $\varphi_b = \neg\ EF(EG\ failure)$ reproduces the same property for s_2, which means that its satisfaction is guaranteed if the property holds in s_2.

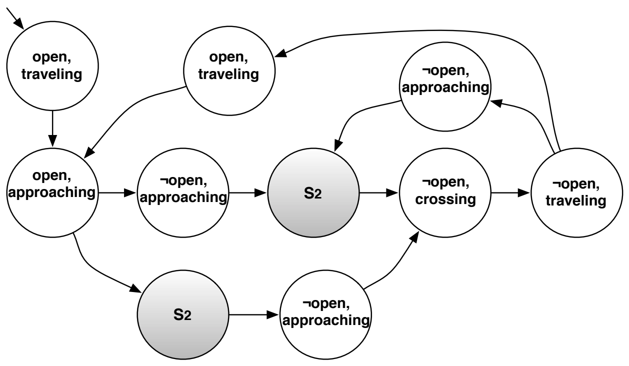


Figure 5.6: *The ILTS for the first refinement of the running example*

State s_2 represents a component that is in charge of controlling the train when it is approaching. In the second refinement, s_2 is elaborated as the second level of the behavior shown in Fig. 5.5. First, the train requests the central station for permission to cross the gate (t_6). If it is granted, event e_4 is generated, and transition t_8 is executed. The system moves to state s_8, from which an acknowledge message is sent to the central station (transition t_9). Instead, if the central station does not grant the permission before a timeout, the system moves to the state s_9, a transparent state that will be later refined. One may say that the train has to stop until the permission is issued by the central station, but it could be an invalid assumption. Thus the refinement is postponed to further requirements elicitations.

Once the second level of the Statechart in Fig. 5.5 is specified, we check whether it satisfies the system requirements: φ'_a and φ_b. AGAVE does not check again the whole Statechart, but checks only s_2 against the derived constraints. The verification of φ'_{a1} and φ'_{a2} against the corresponding ILTS of s_2 (shown in Fig. 5.7) results in reproducing respectively the same properties for s_9. φ'_a requires both of these properties to be satisfied. On the contrary, the verification of the reliability property $\varphi_b = \neg\ EF(EG\ failure)$ returns false because there is the possibility that the system infinitely remains in the failure mode.

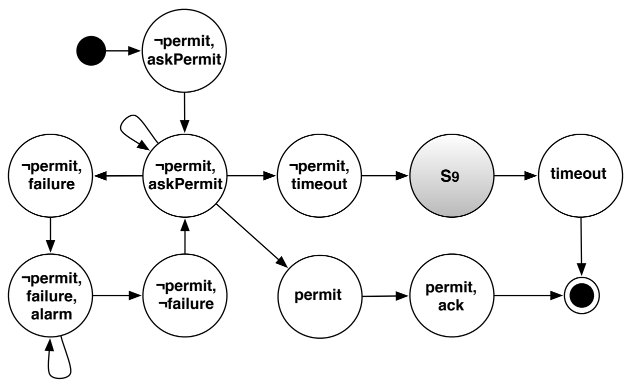


Figure 5.7: *The ILTS for the second refinement of the running example*

5.5 Experimental Evaluation

AGAVE is supported by a prototype verification tool, as a Java standalone application [4]. The tool takes as input two XML files, one representing the model of the system (a Statechart) and one representing the property to verify (in Path-CTL). The tool supports the syntax and semantics described in Section 5.2, and follows the steps of the algorithm presented in Section 5.3. The output of each verification task is either "true", "false", or

[4]Available at `https://sites.google.com/site/amirsharifloo/tool-agave`.

"conditional". In the conditional case, a set of constraints on transparent states is reported as well.

To grasp a better understanding of how the incremental approach is supported by AGAVE can reduce the verification time, we provide the result of an experiment. We assumed that RCS example (Section 5.4) is extended through more refinement steps by providing specifications for each transparent state. We use the first level of the RCS Statechart introduced as refinement at each stage. All the experiments have been carried out on a machine with the following characteristics: CPU 2.53GHz, RAM 4GB, and operating system Windows 7.

Figure 5.8 shows the total time required to verify the new models introduced at each refinement. To demonstrate the advantage of the incremental approach, Figure 5.8 also shows the time needed for the traditional verification (one may call $integrative$), which considers the whole integrated specification at the end of each refinement. As the number of the refinement levels increases, the time of traditional approach rapidly grows (note that the scale is logarithmic). In particular, it reaches 1,93 minutes when a six-level Statechart is analyzed, while AGAVE requires only 328 milliseconds. The main reason of this difference is that the traditional approach at each step verifies the whole integrated specification, while AGAVE only checks the sub-Statecharts of the composite states against the constraints generated by the previous verifications (Figure 5.9). Notice that our tool is a prototype and the result can be improved by applying further optimizations, which will be the goal of future work.

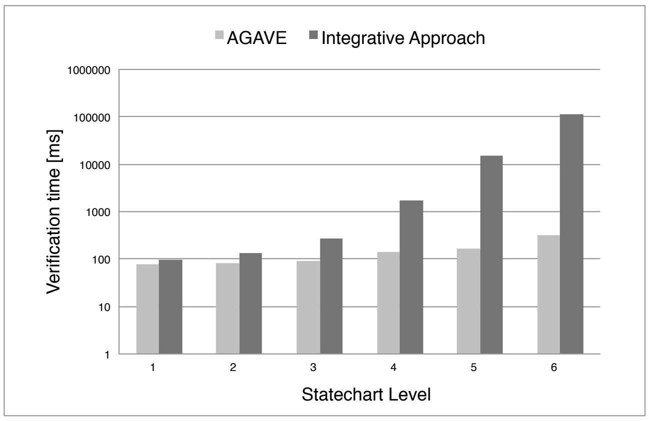


Figure 5.8: *The verification time required for each level of the Statechart*

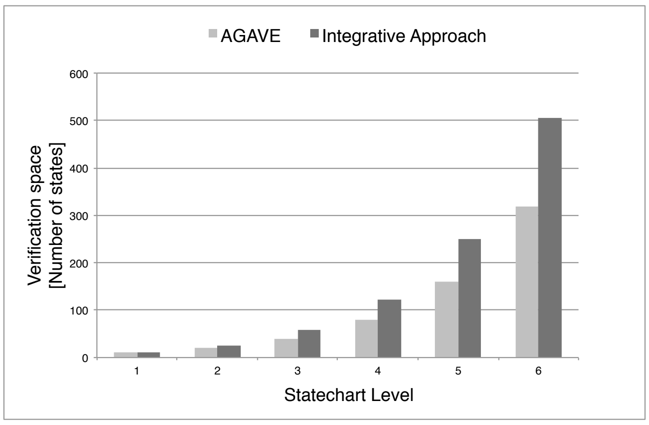


Figure 5.9: *The number of states analyzed for each level of the Statechart*

5.6 Related work

There have been many work addressing the verification of Statecharts [24, 35, 73, 92]. However, to the best of our knowledge, there is no work in the literature dealing with verification of incomplete specification of Statecharts. Zhao and Krogh [92] discuss the verification of CTL properties by introducing *Statechart Kripke Structure* as the underlying analyzable model that is derived from Statecharts. The states are marked by considering the values of variables. The approach explores all execution paths and applies MATLAB Simulink for the analysis.

There exist approaches to transforming Statecharts to hierarchical automata and then applying different verification techniques [24, 35]. Gnesi et al. [35] present an approach through which Statecharts are mapped on hierarchical automata and then a model checking environment, called Jack, is used to check ACTL properties. Dong et al. [24] transform Statecharts into *Extended Hierarchical Automaton* (EHA) to check *Linear-time Temporal Logic* (LTL) properties.

Clark et al. [17] present an approach to verifying Statecharts by using SMV. Although the approach uses the notion of modules in SMV to have a straight-forward mapping, the analysis does not take into account any compositional or reusable verification. Alur and Yannakakis [2] study the verification reuse of hierarchical state machines to avoid re-doing the whole verification when some state machines change. This approach has similarities with AGAVE; however, they take a bottom-up approach and calculate the verification results starting from the lately-specified components that show up at the end of refinements. On the contrary, AGAVE can be viewed as a kind of assume-guarantee method [46], [47], [70], applied to high-level models and incomplete refinements.

However, the existing assume-guarantee methods view a system as a collection of cooperating components, each of which has to guarantee certain properties. A component is verified independently from the others assuming a certain behavior of the components it interacts with [70]. Differently our approach considers the whole specification as the system behavior and employes the model-checking techniques, which produce constraints for unspecified components.

5.7 Conclusion

This chapter presents AGAVE, a methodology that aims to support developers in the iterative and incremental system specification. AGAVE offers a technique to verify temporal logic properties of incomplete models, expressed in state-based formalisms, where some parts are unspecified. These unspecified parts may be later refined and elaborated during the development or left as components dynamically deployable at run time.

The methodology exploits the hierarchy of the model, applying the verification iteratively. At each stage, a set of properties are deduced that have to be satisfied by the refinements to guarantee the satisfaction of the original property. AGAVE has been implemented for Statecharts and Path-CTL temporal logic. We have evaluated it through a case study and demonstrated the benefits of such approach by performing state space and performance analysis.

Part III

Stochastic Modeling and Verification of Closed Variability

CHAPTER 6

Modeling and Verification of Stochastic Software Product Lines

"*Give me a place to stand and with a lever I will move the whole world.*"
Archimedes

Engineering SPLs has been of interest for both industry and academical research. The advantage of establishing an SPL and producing variety of products, that share a common set of features, has been proved by modern companies [1]. A development approach inspired by SPL principles aims at improving productivity and reducing the time and cost to develop a family of products. *Reusability* is the key point to achieve such benefits through different software engineering disciplines from requirements engineering [63] to implementation and testing [53].

Devising model-checking techniques for SPLs is of importance, due to the fact that a design flaw is propagated to many products. To reach such aim, the first step is to represent an SPL specification via a compact and user-friendly notation, instead of individual modeling of each prod-

[1]Product Line Hall of Fame: `http://splc.net/fame.html`

uct. Such representation shall come with a clear semantics, which would allow us to reason about product behaviors. Classen et al. [20] took the initial steps by introducing FTS as an extension of transition systems. They expanded their work by providing model-checking techniques capable of verifying all products at once for LTL and CTL properties [19]. Although not so formal, there exist several proposals to capture behaviors of SPLs by using high-level models such UML diagrams [38, 93]. High-level models provide more intuitive representations and simplify understanding and communication among stakeholders. The succinctness is the main advantage of such modeling approach w.r.t. FTS.

In this chapter, we take a further step in modeling SPL behaviors and formally verifying their properties. In contrast to the existing techniques, we focus on stochastic properties, which are more appropriate for specifying non-functional requirements (hereafter *NFRs*). In particular we describe how to model variability and stochasticity in UML Sequence Diagrams (SDs). Walking through our running example, the classic example of SPL vending machine is represented by UML SDs and Feature Diagrams (FDs). Regarding NFRs, our focus is on reliability and energy consumption. We discuss how SDs annotated with stochastic information are used to reason about the satisfaction of these two NFRs.

Figure 6.1 illustrates our framework. As shown, the framework consists of both modeling and verification of NFRs for SPLs. It is described how stochastic SPLs are modeled by SDs, and how they are transformed into a new variation of Markov models, which are suitable for model checking. In this direction, we introduce Featured Discrete-Time Markov Chains (FDTMCs), and later show how they can be model-checked for predicting NFRs satisfaction of all products of an SPL. In the following sections, we first briefly overview our running example, and then present the modeling, transformation, and verification techniques employed within the framework.

6.1 Motivating Running Example

As a running example, we recall the SPL of the vending machine earlier introduced in Chapter 2. The example refers to a product line that produces beverage vending machines. Figure 2.1 shows the feature model for the product line. The general scenario starts by inserting coins and receiving changes by the user. Then she chooses a beverage, which is consequently prepared and delivered by the vending machine. As illustrated in the feature model, a vending machine may have the capability to prepare tea, soda, or

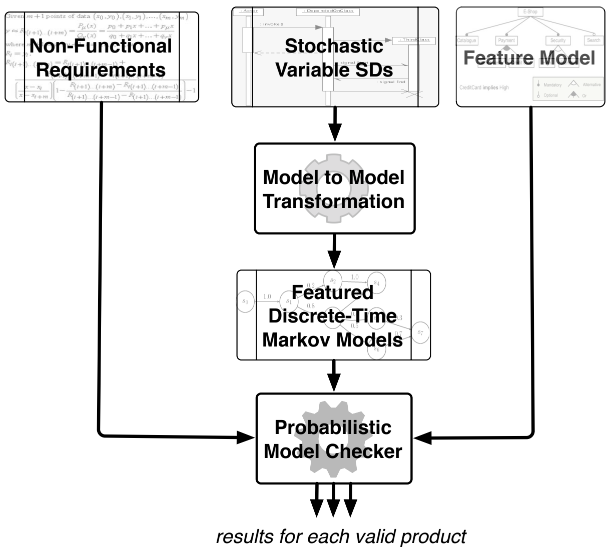

Non-Functional Requirements
Stochastic Variable SDs
Feature Model
Model to Model Transformation
Featured Discrete-Time Markov Models
Probabilistic Model Checker
results for each valid product

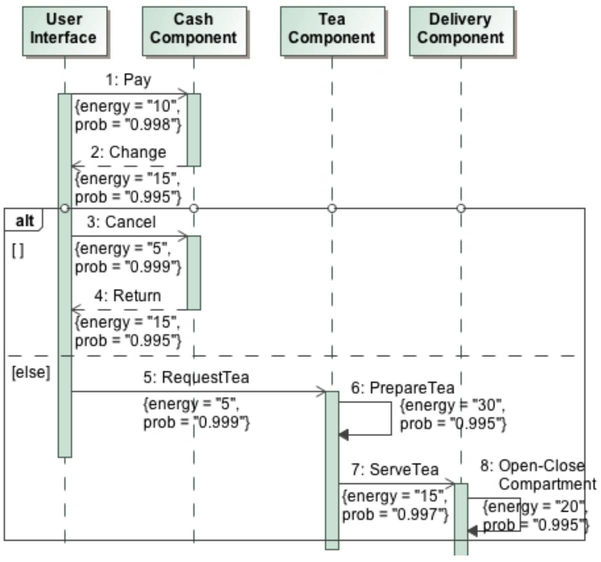
Figure 6.1: *The Variable SD for the whole product line of the vending machine*

both. It may require a cash payment or may simply offer a free drink. Similarly, it may support a feature to add flavor.

Different vending machines can be derived from this feature model. For example, one may support *tea* and *cash*. The other may offer the features *soda*, *free* and *taste*. Figures 6.2 and 6.3 show the high-level interactions of system components for these two products. As illustrated in Figure 6.2, each action is represented by a message, and is annotated with reliability and energy consumption values. Initially, the user inserts coins and then receives the change. Message *Pay* shows the act of inserting coins and receiving the change. Regarding the annotations, the message is annotated with *prob="0.998"*, which means that the action of payment is successfully performed with a probability of 0.998. Moreover, the message is annotated with *energy="10"*, which means 10 "units of energy" are consumed to accomplish this action. Then the system calculates the difference between the money inserted and the product's price, and returns it as a change to the user. The message *Return* represents this action. The scenario continues with two possible choices taken by the user. She may cancel the order or may choose *tea*, which is followed by a set of actions eventually leading to beverage delivery.

As mentioned earlier, we are interested in specifying and verifying reliability and energy consumption properties of each product. For example, an

Figure 6.2: *Product 1 (*cash *and* tea *)*

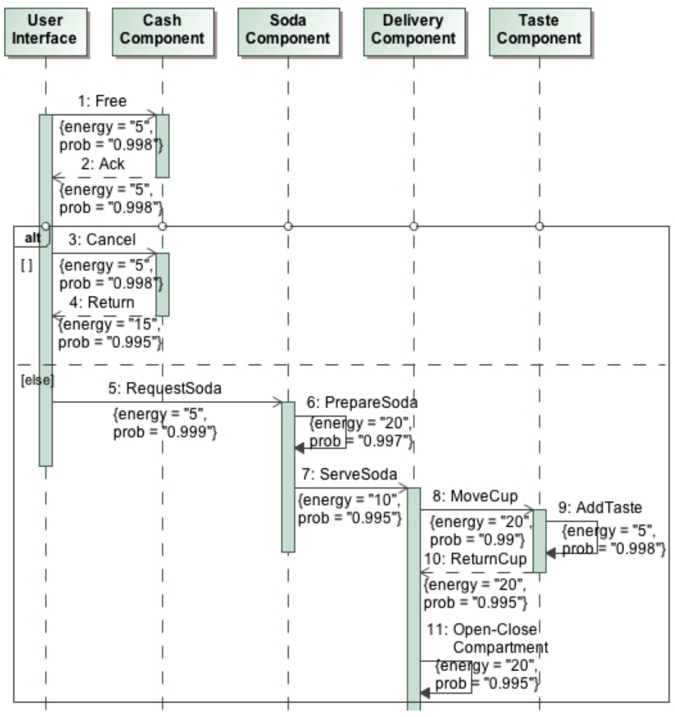

User Interface
Cash Component
Tea Component
Delivery Component
1: Pay
{energy = "10", prob = "0.998"}
2: Change
{energy = "15", prob = "0.995"}
alt
[]
3: Cancel
{energy = "5", prob = "0.999"}
4: Return
{energy = "15", prob = "0.995"}
[else]
5: RequestTea
{energy = "5", prob = "0.999"}
6: PrepareTea
{energy = "30", prob = "0.995"}
7: ServeTea
{energy = "15", prob = "0.997"}
8: Open-Close Compartment
{energy = "20", prob = "0.995"}

important property is the average energy consumption of a specific instance of the vending machine. Similarly, one might be interested in knowing how reliable the functionality of beverage delivery is. Later, we describe how to formally specify and verify such properties for all products derived from an SPL.

6.2 Stochastic Variable Sequence Diagrams

To capture the behavior of an SPL, we augment UML SDs with variability. Hereafter the resulting diagrams will be called *Variable SDs*. The behavioral variability is identified and modeled as variation points and alternatives within this kind of SDs. Variable SDs consist of a set of main elements of UML SDs and a set of stereotyped elements to specify variability. The notation supports the following building blocks: *lifelines*, *messages*, and *combined fragments*. It also supports the following types of messages: *synchronous*, *asynchronous*, and *reply*; they are indicated by a line with solid arrowhead, a line with an open arrowhead, and a dashed line with an open arrowhead, respectively. Messages play a major role; they are used to represent both communication and computation. We use *self-messages* to indicate the execution of internal actions by a lifeline, while inter-object messages stand for a communication. Figure 6.4(a) demonstrates these var-

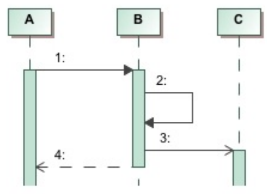
Figure 6.3: *Product 2 (*free*,* soda*, and* taste*)*


User Interface
Cash Component
Soda Component
Delivery Component
Taste Component
1: Free
{energy = "5", prob = "0.998"}
2: Ack
{energy = "5", prob = "0.998"}
alt
[ ]
3: Cancel
{energy = "5", prob = "0.998"}
4: Return
{energy = "15", prob = "0.995"}
[else]
5: RequestSoda
{energy = "5", prob = "0.999"}
6: PrepareSoda
{energy = "20", prob = "0.997"}
7: ServeSoda
{energy = "10", prob = "0.995"}
8: MoveCup
{energy = "20", prob = "0.99"}
9: AddTaste
{energy = "5", prob = "0.998"}
10: ReturnCup
{energy = "20", prob = "0.995"}
11: Open-Close Compartment
{energy = "20", prob = "0.995"}

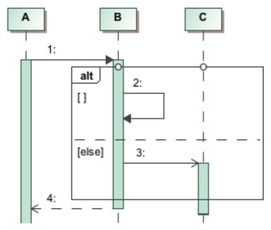
ious message types. Notice that if an object sends a synchronous message, it remains blocked until it receives a reply message. Instead, when an object sends an asynchronous message, it continues with the rest of the actions it is expected to perform. A reply message is a kind of asynchronous message, as far as the issuer is concerned.

Messages are transmitted according to partial order semantics, which includes two rules: (1) the actions of a lifeline are ordered from top to down, (2) a message cannot be received before it is sent. Sets of messages can be grouped together in combined fragments, graphically represented by a box. The official specification of UML comprises many different types of combined fragments. Hereafter we focus on the four major fragments: (1) Alternative, (2) Option, (3) Loop, and (4) Parallel.

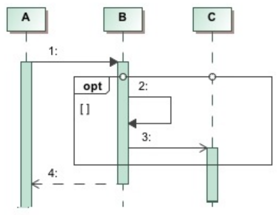
Mutually exclusive choices between two or more sequences of messages are represented using *Alternative*. Each Alternative contains a set of operands (each of which is a group of messages) separated by a dashed line. Each operand is associated with a condition and is executed if the condition evaluates to true (Figure 6.4(b)). Note that the condition of Alter-



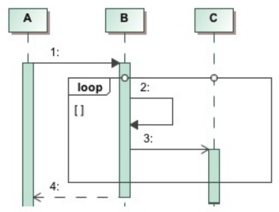
(a) Simple diagram



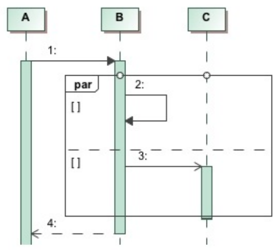
(b) Alternative



(c) Option



(d) Loop



(e) Parallel

Figure 6.4: *Sequence diagrams*

native is evaluated only once when all the lifelines participating in it have reached the Alternative. In fact, we follow the proposal by Alur [2] and apply a synchronous approach, which is also used for the other fragments (Option and Loop) whose execution is conditional.

Option fragment is used to model a sequence that occurs if and only if a certain condition evaluates to true (Figure 6.4(c)). SDs represent iterating sequences of messages through loops. A *Loop* is associated with a condition and the sequence of messages included in the fragment is executed as long as the condition evaluates to true (Figure 6.4(d)). The *Parallel* fragment represents parallel computations (Figure 6.4(e)).

As aforementioned, Variable SDs are able to capture the variability in a behavior. To do that, SDs are augmented with variation points and alternatives which are linked to the elements of the SPL feature diagram. More precisely, variation points are represented by stereotyped Option or Alternative fragments. The operands of those fragments contain the alternative behaviors for variation points. Each variation point is linked to a feature in the SPL feature diagram, which shares the same name. The relationships in the feature model define the semantics of variation points in the Variable SD. An Alternative fragment is used to show an OR relation between a feature and its children. Similarly, an Option fragment is used to illustrate an optional feature. These fragments are discriminated with tags on the top-right side. Options are tagged with the keyword *Optional point*, and Alternatives are tagged with *Variation point*.

Figure 6.5 shows the Variable SD of the running example. As shown, the Option fragment *Taste* is an optional feature. Alternative fragments are used to model variation points in which we may select a feature among several of them. An Alternative may represent an inclusive OR, which means that at least one of the operands is included in any derived product. The Alternative tagged with "Beverage" is an example for this kind. An Alternative may also support an exclusive OR semantics, which means that one and only one of the operands can be included in a product. "Payment" feature is an example of this kind.

For the sake of stochasticity modeling and quality analysis, the elements of SDs are annotated by MARTE profile [62]. We annotate all messages of a diagram with two MARTE properties: *energy* and *prob*. The former –a non-negative real number– represents the energy needed for message transmission. The latter represents the probability that a message transmission is successfully performed. Note that since self-messages represent internal operation, these properties represent the energy consumption and the reliability of the operation, respectively.

Combined fragments, except Parallel and those dedicated to capturing variability, are annotated with execution probabilities. The probabilities are added to resolve the non-determinism introduced by these fragments. A probability attached to an Option indicates the likelihood that the optional behavior is performed during a random execution trace. Similarly, each loop is annotated with a probability, which expresses the probability that the loop may iterate. Since Alternative fragment includes more than one operand, each operand is annotated with an execution probability [2].

Variable SDs provide a unified representation for all products of an SPL; however, they are not suitable for formal analysis of requirements. For this reason, we transform them into FDTMCs, which are later analyzed through model checking.

6.3 Featured Discrete-Time Markov Chains (FDTMCs)

To capture variability in SPLs we define Featured DTMCs (Hereafter *FDTMC*) as extension of DTMCs, which allow us to represent different probability

[2]The sum of the probabilities for an Alternative fragment must be always 1.0.

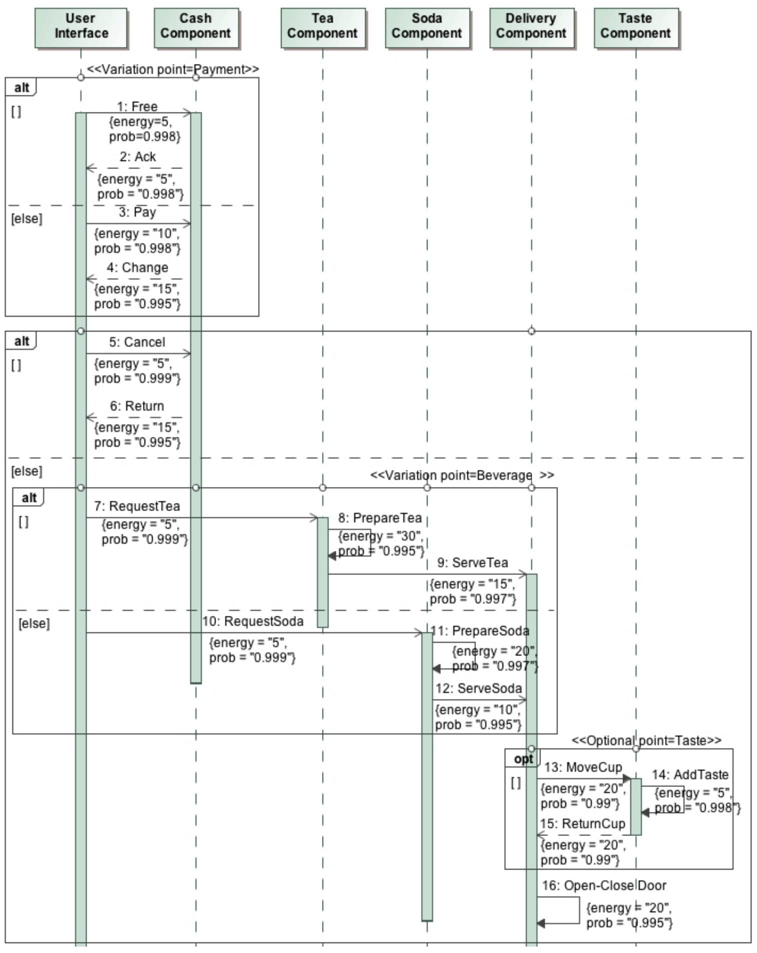

User Interface
Cash Component
Tea Component
Soda Component
Delivery Component
Taste Component
<<Variation point=Payment>>
alt
[]
1: Free {energy=5, prob=0.998}
2: Ack {energy = "5", prob = "0.998"}
[else]
3: Pay {energy = "10", prob = "0.998"}
4: Change {energy = "15", prob = "0.995"}
alt
[]
5: Cancel {energy = "5", prob = "0.999"}
6: Return {energy = "15", prob = "0.995"}
[else]
<<Variation point=Beverage >>
alt
[]
7: RequestTea {energy = "5", prob = "0.999"}
8: PrepareTea {energy = "30", prob = "0.995"}
9: ServeTea {energy = "15", prob = "0.997"}
[else]
10: RequestSoda {energy = "5", prob = "0.999"}
11: PrepareSoda {energy = "20", prob = "0.997"}
12: ServeSoda {energy = "10", prob = "0.995"}
<<Optional point=Taste>>
opt
[]
13: MoveCup {energy = "20", prob = "0.99"}
14: AddTaste {energy = "5", prob = "0.998"}
15: ReturnCup {energy = "20", prob = "0.99"}
16: Open-Close Door {energy = "20", prob = "0.995"}

Figure 6.5: *The Variable SD for the whole product line of the vending machine*

distributions over each transition. More precisely, the probability distribution of a transition varies depending on the enabled features. In other words, the probability distribution is a function of features, which themselves are boolean values. An FDTMC is defined as a tuple $(S, \nu, d, \Pi, P, \Xi, W)$ where

- S is a countable, non-empty set of states;
- ν is a vector of size $|S|$ that records the initial probability distribution of every state;
- d is a feature diagram;

- $\Pi : F_d \rightarrow [0, 1]$ is a total function called probability profile;
- $P : S \times S \rightarrow \Pi$ is the transition probability function, which assigns a probability value to every transition;
- $\Xi : F_d \rightarrow \mathbb{R}^+_{\not{0}}$ is a total function called reward profile;
- $W : S \times S \rightarrow \Xi$ is the transition reward function, which assigns a reward value to every transition;

where F_d is a feature expression that is valid w.r.t feature diagram d. Note that any instance of FDTMC must ensure the satisfaction of the usual *probability axioms*. In particular, the sum of the probability of each random outcome must be equal to one. It implies that the sum of the initial probability of all the states must be one : $\sum_{s \in S} \nu(s) = 1$. For every state, the probability sum of its outgoing transition must be one as well. Since features can influence transition probabilities, this assertion must hold for all the valid products. The actual value of a product is thus determined according to the considered product and the probability profile Π.

6.4 Model to Model Transformation

This section explains how StochasticVariable SDs (hereafter SVSDs) are transformed into FDTMCs. Given an SVSD, we generate two FDTMCs: one representing reliability model and another for energy consumption. The procedure of both transformations are the same with only difference in mapping messages. As for reliability model, each message is transformed in two transitions to represent both success and failure. Regarding an energy model, a message is transformed into only one transition annotated with energy profile. In the following, we first describe the procedure for reliability models and then discuss its variation for energy consumption.

To apply the transformation, first we need to find the order in which actions are performed (which in our case corresponds to finding the order in which messages are transmitted) by iteratively performing a search in the diagram to find performable actions. We start in an initial state in which lifelines are initiated, and then we find the next performable actions (self-message or message) and transform them into transitions of target FDTMC.

Algorithm 1 illustrates the algorithm to extract performable actions. For simplicity, let us assume that the SVSD does not include any combined fragment. To find the actions, the method goes through each lifeline and checks the upcoming actions that the lifeline is supposed to perform. If the action is a self-message, it is performable without any condition. In case it

Algorithm 1 Retrieving performable actions

```
RetrievePerformableActions($SD\ sd$){
  var $Action[]\ \ actions$;
  foreach ($lifeline[i] \in sd.lifelines$){
    if ($\neg lifeline[i].IsFailed()$ & &$\neg lifeline[i].IsFinished()$)
      $actions.add(RetrievePerformableActions(lifeline[i]))$;
  }
}
RetrievePerformableActions($Lifeline\ l$){
  var $Action[]\ actions$;
  while (){
    switch ($l.current_action.type$){
      case $"RECEIVE"$ :
        if($\neg IsAlreadySent(l.current_action)$){
          return $actions$;
        else;
          $l.moveToNext()$;;
      case $"SEND-ASYN"$ :
        $actions.add(l.current_action)$
        $l.moveToNext()$)
      case $"SELF-MSG"$ || $"SEND-SYNC"$ :
        $actions.add(l.current_action)$;
        return $actions$;
    }
  }
}
```

is a receive action, it is necessary to check whether the sender lifeline has transmitted the message. If it has done so, receiving is performed and also the next action is checked. If the action sends a message, the type of message is considered. If it is synchronous, it is added as a performable action. If it is asynchronous, not only it is added to the list of performable actions, but also the next performable action sought. The reason is that sending an asynchronous message does not block a lifeline, so it can continue immediately with the next action.

Transforming messages to transitions depend on the type of properties that one would like to check. In case of a reliability property, the actions on the SVSD are annotated with their success probability P ($1 - P$ is the probability of failure). Algorithm 2 describes how to map an SD onto a FDTMC. Method *Transform* invokes method RetrievePerformableAction (discussed earlier) to extract actions given the execution locations of each lifeline. Each action is transformed into two transitions, representing success and failure. If an action is performed correctly, the corresponding lifeline in the SVSD moves to the next action and a success transition is added to the FDTMC. Otherwise, the receiver lifeline involved in the action is failed and a failure transition is added (Figure 6.6(a)). In Algorithm 2, the methods *Create_Success* and *Create_Failure* are responsible for adding these transitions to FDTMC.

After performing any action, the FDTMC transitions from the current state to a new state. If there is more than one performable action, we add new transitions to represent the interleaving. For instance, Figure 6.6(b) shows that N transitions are added to the FDTMC. Each of these transitions assumes that one particular action is performed before the others. The probability of each transition equals to *1/N*, N being the number of performable actions.

To simplify the exposition, Algorithms 1 and 2 describe the method to find and transform performable actions of a diagram without combined fragments. To cope with these fragments, we follow the rules described below:

Option. If the action we are considering is the first action of an Option fragment, two different traces may exist. The first trace occurs if the condition of the Option is true (with probability *Opt*) and its actions are performed. On the other hand, the second trace occurs if the condition is evaluated to false (with probability of *1-Opt*) and the actions are ignored. In the latter case, each lifeline involved in the Option moves to the location after the Option, and performable actions are retrieved from there. In the corresponding FDTMC, two new transitions and states are added as shown in Figure 6.6(c).

Alternative. Transforming an Alternative fragment is quite similar to an Option, with the difference that one of the operands is chosen and its actions are performed. The probability attached to each transition *op1*, .. , *opN* (Figure 6.6(d)) is the probability that the operand is chosen.

Loop. If the first action of a Loop is visited, a strategy similar to Option is used to map the first iteration. However, when the last action of the Loop is performed, two traces may be executed depending on the condition of Loop: either Loop is iterated once more or it is exited (Figure 6.6(e)).

Parallel. In case an action is the first action of a Parallel fragment, all its operands are activated and shall be performed in a parallel manner. To cope with such situation, we consider all interleavings between the actions of the operands.

The mapping of variation points expressed through Alternative and Option is similar to the above; however, instead the probabilities over the

Algorithm 2 Ttransforming Variable SD to FDTMC

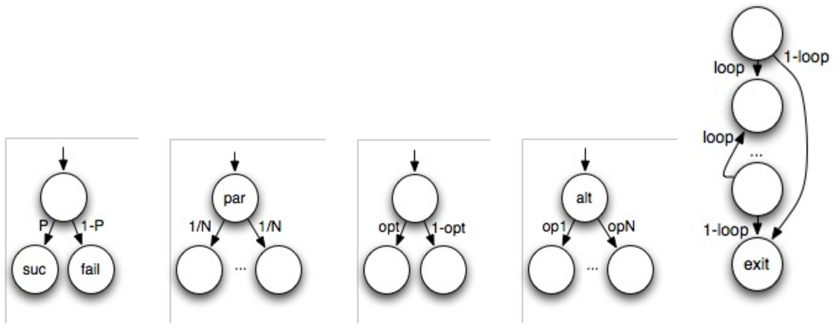
```
TransformSDtoDTMC($Lifeline\ l$){
 var $FDTMC\ dtmc$;
 var $State\ initial_State$;
 var $Action[]\ actions$;
 $dtmc.add(initial_State)$;
 $Transform(sd,\ dtmc,\ initial\_state,\ actions)$;
}
Transform($SD\ sd,\ FDTMC\ dtmc,\ State\ current\_state,\ Action[]\ actions$){
 while ($\neg sd.IsFinished()$){
 $actions.add(RetrievePerformableActions(sd))$;
  if ($actions.size == 0$)
   return; //the procedure terminates
  if ($actions.size == 1$){
   $action = actions.get(0)$;;
   $Create\_Success(current\_state,\ action,\ actions,\ sd,\ dtmc)$;
   $Create\_Failure(current\_state,\ action,\ actions,\ sd,\ dtmc)$;
  }else if ($actions.size > 1$){
   var $float\ branchProb\ =\ 1/actions.size$;
   foreach($action \in actions$){
    var $State\ middle\_state\ =\ new\ State$;
   $middle\_tra = new\ Transition(current\_state,\ middle\_state,\ branchProb)$;
    $dtmc.add(middle\_state)$;
    $Create\_Success(current\_state,\ action,\ actions,\ sd,\ dtmc)$;
    $Create\_Failure(current\_state,\ action,\ actions,\ sd,\ dtmc)$;
    $dtmc.add(middle\_tra)$};
  }
 }
}
```

added transitions are replaced with probability profiles. The probability profile for an Option fragment with a feature expression f_x is shown in the below.

$$\Pi_x = (f_x : 1) or (\neg f_x : 0) \tag{6.1}$$

This profile is assigned to the transition entering the first state of the Option. Similarly the transition jumping over the Option behavior is augmented with a negated profile, as shown below. Note that given a product only one of the profile returns one, and so the sum of both transitions is always equal to one.

$$\Pi_{\neg x} = (f_x : 0) or (\neg f_x : 1) \tag{6.2}$$



(a) Message (b) Interleaving (c) Option (d) Alternative (e) Loop

Figure 6.6: *Transformation rules for DTMC*

In principle, Option is a special case of Alternative, when there is only one alternative behavior. We extend the above approach to the transitions produced for an Alternative fragment. Considering an Alternative with N alternating behaviors, the probability profile for each behavior is as the below. Note that in case of an inclusive Alternative, where more than one operand is enabled, the probability of each operand is equal. Regarding an exclusive Alternative, only one of the operand is enabled which would take probability of one.

$$\Pi_x = (f_x : \frac{f_x}{\sum_N f_i}) or (\neg f_x : 0) \tag{6.3}$$

Figure 6.7 shows the FDTMC of the reliability model for our SPL vending machine. An SVSD is transformed into a FDTMC to verify energy consumption properties specified in Reward logic. This transformation is very similar to the transformation we described for reliability models. On the contrary, each message is transformed into a transition annotated with a reward profile which specifies the energy consumption of an operation. Note that there is no failure or success transition in the target models, because they are produced only for reliability analysis. The number of the transitions in the target models depends on the complexity of the behavior. If the behavior contains many interleavings, alternative branches and loops, then the corresponding FDTMC would have a complex structure with many transitions.

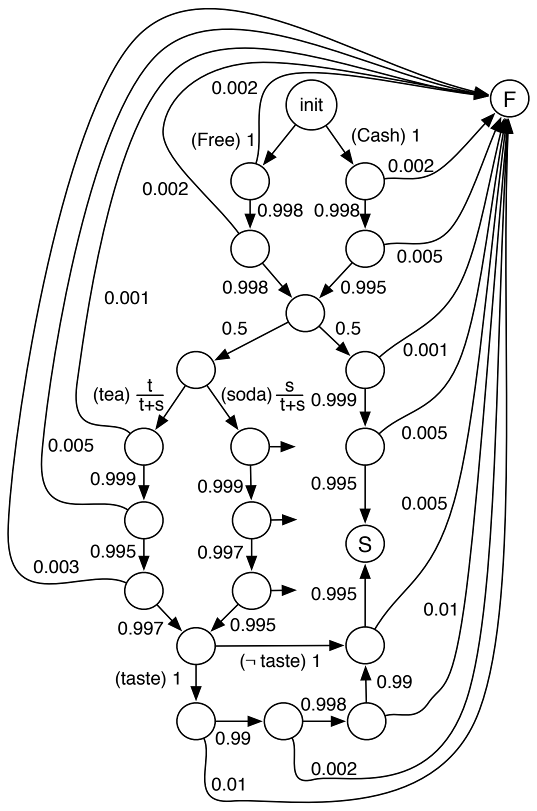


Figure 6.7: *The FDTMC for the vending machine SPL*

6.5 FDTMC Model Checking

Here we discuss how to verify a FDTMC against PCTL and Reward logic, which paves the way to check reliability and energy consumption properties for all products of a SPL. We present three verification approaches, each of which is efficient for a certain kind of SPLs.

6.5.1 Enumerative Verification

This procedure, called *enumerative*, consists of projecting FDTMC to Markov models of each product, and then model checking each projection individually, where the number of Markov models is equal to the number of products. Regarding reliability properties, FDTMC is projected to DTMCs by simply providing the enabled features of each product. This information can be derived from the associated feature diagram, which guarantees only the DTMCs of valid products are generated. Having the DTMCs produced, we model-check each of them against the reliability property by using a probabilistic model checker (e.g. PRISM). As for energy consumption analysis, we project FDTMC to DTMCs with Rewards and apply the probabilistic model checker fed by energy requirements expressed in Reward logic. Although this method takes the advantage of efficient existing model checkers, it clearly ignores the commonalities between the products. Given a large number of products, numerous DTMCs would be generated. To handle such cases, we propose another approach based on *parametric* model checking of Markov models.

6.5.2 Parametric Verification

We can convert an FDTMC into a parametric DTMC (similarly a parametric DTMCs with Rewards) where the parameters are the features, and then can reuse existing methods for model-checking parametric DTMCs. Converting probability profiles to parametric expressions is the crux of this approach. To do so, we represent the function $\Pi(\alpha)$ as the parametric expression obtained from probability profile α:

$$\sum_{p_i \in \mathcal{P}(N)} \epsilon(p_i) \times \Pi(\alpha, p_i) \tag{6.4}$$

Regarding a particular product p, every term of this sum except one is equal to 0; the remaining term is equivalent to $\Pi(\alpha, p)$. Note that if several products share the same value for α, we can drastically simplify the above sum. In particular, if the value of α depends only on a feature f, then we can rewrite Equation 6.4 as $f \times \alpha_1 + (1 - f) \times \alpha_0$ where α_1 (resp. α_0) is the value of α when the feature f is enabled (resp. disabled). The second form of variability makes the reward earned through the execution of a transition depend on features as well. To cope with it, we use a technique similar to the previous one, that is, we define the reward of a given transition as a parametric expression over the set of features, that is, we represent $\Xi(s)$ by the expression:

$$\sum_{p_i \in \mathcal{P}(N)} \epsilon(p_i) + \Xi(\alpha, p_i) \tag{6.5}$$

As for the previous case, this sum will more likely appear in a simplified form in practice. By performing the above two transformations, we reduce FDTMC verification to parametric DTMC verification, where the parameters are all Boolean variables that model the presence or absence of features.

In consequence, we are able to use existing parametric model-checkers like PARAM [40] and the tool of Filieri et al. [32]. Given a parametric model, they return a rational expression containing parameters. In our context, the parameters are variables corresponding to the features. The expression represents the probability or the reward we seek. To determine the probability or the reward of a given product, we simply replace, in the expression, each feature by 1 if it belongs to the product, or by 0 if it does not. Then, we obtain an actual real value.

6.5.3 Approximative Verification

In this section, we present an approximative approach to checking stochastic properties of FDTMCs. The motivation behind this approach is to find an approximative result within a time limit. First we describe the algorithm to deal with PCTL and then Reward logic.

PCTL Verification

Let us first recall PCTL property language. PCTL is defined by the following syntax:

$$\Phi ::= true \mid a \mid \Phi \ \wedge \ \Phi \mid \neg \ \Phi \mid \mathcal{P}_{\bowtie p} \left(\Psi\right)$$
$$\Psi ::= X\Phi \mid \Phi_1 U^{\leq t}\Phi_2 | \ \Phi_1 U \Phi_2$$

where $p \in [0,1]$, $\bowtie \in \{<, \leq, >, \geq\}$, $t \in \mathcal{N} \cup \{\infty\}$, and a represents an atomic proposition. Since $U^{\leq t}$ can represent Next operator by considering ($t = 1$), and infinite Until by ($t = \infty$), we focus on the verification of bounded Until formulae and consider the others as particular cases.

The algorithm first starts with parsing the property, and building the parsing tree, in which the leaves are atomic propositions and the inner nodes are the operators. Similarly to CTL model checking [4], the set of satisfying states is assigned to each node of the tree through a bottom-up evaluation. Differently from CTL, a probability satisfaction is also calculated for each state. While the atomic propositions and boolean operators are treated as in CTL verification, we apply a different way to deal with Until operator. To tackle the time limit, we set two parameters: the path length and the maximum number of considered paths, by which the algorithm can be guided to explore the state space. This way, we enforce the algorithm to check the property based on a limited number of paths, which are shorter than a threshold.

Let us consider the case in which each transition profile in FDTMC contains only a probability and a constraining feature expression. Given

$\phi_1 U^{\leq} \phi_2$, a bounded search is performed to find a set of paths, with the length less or equal t, on which all the states satisfy ϕ_1 and the last states satisfy ϕ_2. We call the probability that a certain path is executed *path probability*, which is calculated by the product of the probabilities of its comprising transitions. Moreover, every path is constrained with a *path constraint*, which is a boolean expression obtained as the conjunction of the feature expressions of its comprising transitions. Given a state s, the probability of having the formula satisfied equals to the sum of the path probabilities of the set of paths, which start from s and end in a state satisfying ϕ_2[3]. The property is constrained by the disjunction of the path constraints.

Considering FDTMCs with profiles, the procedure is the same except that instead of a single probability/constraint a *path profile* is calculated for each path. Each profile contains a set of probability/constraint elements, which are calculated by the cartesian products of all transition profiles visited over the path. Similarly, a profile is calculated for a property satisfaction given an initial state.

Reward Logic Verification

The approach to verifying Reward properties of an FDTMC is similar to PCTL verification but deals with calculating rewards. Here we focus on reachability reward properties, though our approach is simply extendible to other reward operators. The reachability reward is expressed as below, where $\sim r$ represents a comparison relation ($\leq r$, $\geq r$, and $= r$) in which r is a real number.

$$R_{\sim r}[F\phi]$$

$R_{\sim r}[F\phi]$ represents a reachability property for a path starting from an initial state and ending in states where ϕ holds. The calculated reward is compared with r, and so the result is either True or False. If r is replaced with a question mark, then the calculated reward will be returned instead. To calculate the reward, similarly to PCTL verification, we generate a set of paths regarding a time limit. Given a set of paths, we calculate the profile that contains a set of reward values, each of which constrained by a feature expression. A path reward is calculated by accumulating the reward obtained in each state visited on the path.

[3]Note that the satisfaction probability of a state in which ϕ_2 holds is *one*.

6.6 Experiments

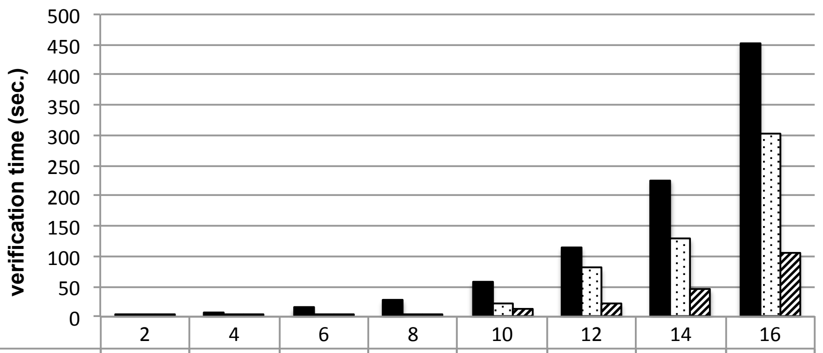
In this section, we report the results obtained by evaluating the performance of the three FDTMC verification techniques in terms of verification time. We consider two technical case studies as our benchmarks, which we systematically extend to obtain larger models. The first case study is an abstract model of failure-recovery systems, in which the system goes through successive degradation states to eventually reach an absorbing failure state. In every degradation state, however, instead of going to the next degradation state, the system may completely break and reach a second absorbing failure state. In every degradation state, the system may also partially recover and reach the previous degradation state. Apart from the initial state and the two absorbing states, the probability of the transitions leaving each state depends on the presence and absence of specific features. The model is extended by adding new degradation states and features.

The second case study is an abstract model of a service provider system that gives the opportunity to its users to invoke different services. The execution of a service is modeled by a sequence of states. During such executions, the system may fail and suddenly reach a failure (absorbing) state. After any service execution, the system may keep executing more services or may go to an absorbing successful-termination node. Each service requires a specific feature to be started, hence the variability within such a system. Unlike the first model, the behavior of the features are completely independent in this model. We enlarge the model by gradually adding new services, which also increases the number of features.

For both examples, we checked the probability with which system reaches a failure state is below 0.1. This reachability property can be expressed in PCTL as $\mathbb{P}_{<0.1}\big(\diamond\, failure)\big)$. All the benchmarks were run on a Dual Intel(R) Xeon(R) CPU E5530 @2.40GHz with 4Gb of RAM. To perform the enumerative verification, all the DTMCs modeling a specific product are first derived from the FDTMC and then verified one-by-one by using the PRISM model checker[4] [52]. For the parametric approach, we use the parametric model checker developed by Filieri et al. [32], and then evaluate the resulting expression by using JEP Java library[5]. For the bounded approach, we use a prototype we developed from scratch.

The total verification time of the enumerative approach is the sum of the model-checking times to verify each single product by PRISM. We excluded the time to produce individual DTMCs as well as the time taken for

[4]http://www.prismmodelchecker.org/
[5]http://www.singularsys.com/jep/



	2	4	6	8	10	12	14	16
■ Enumerative	3,4	7,105	14,564	27,928	57,175	115,027	226,233	452,556
□ Parametric	0,214	0,279	0,386	1,204	20,953	80,443	129,058	302,248
▨ Bounded	0,044	0,078	0,349	1,304	11,645	21,81	44,942	104,915

number of features

Figure 6.8: *The verification time of failure-recovery case study*

creating PRISM input files. As for the parametric approach, the verification time is obtained by summing the parametric verification time and the time spent to evaluate the expression for each single product. Since the bounded approach verifies all products together, its verification time equals to the total time taken by the prototype tool. In all the experiments, we set the bound of the algorithm such that the maximum precision error is always less than 10^{-3}.

Figure 6.8 shows the verification times for the failure-recovery case study. In this case, the number of features f grows from 2 to 16. It turns out that the bounded approach outperforms the others in almost every case. The verification time of the enumerative approach grows exponentially with the number of features, as expected. We also observed that the parametric approach suffers from the growing complexity of the rational function.

Table 6.1 reports the time each of the verification techniques takes to verify the models of the service provider case study. The results show that the enumerative approach takes a longer time, while the other two techniques exhibit a similar performance. In contrast to the first case, the parametric approach outperforms the bounded technique.

The results of our experiments suggest that the enumerative approach is increasingly inefficient as the number of features (and hence of products) increases. Still, if a verification task only deals with a small number of products, the enumerative approach is a reasonable choice. The cost of the parametric approach is highly dependent on the complexity of the

Table 6.1: *The verification time of service provider case study (in seconds)*

Features	Enumerative	Parametric	Bounded
2	4,207	0,237	0,111
4	6,932	0,267	0,306
6	14,57	0,336	0,314
8	29,224	0,408	0,519
10	57,215	0,53	0,676
12	119,544	0,572	1,636
14	227,91	0,99	1,931
16	466,185	1,126	2,966

rational function that is produced by the parametric model checker. This complexity varies depending on the topology of the model. In the first case, where feature-dependent transitions occur sequentially, the verification time grows faster than in the second case, where the features are scattered around the model. The approximative algorithm exhibits a good performance in both experiments.

Our theory is that the parametric approach performs better in models where feature-dependent transitions do often not occur in sequence, that is, when there is a limited number of *feature interactions*. On the contrary, if many of these sequences occur in the model then the size of the function returned by the parametric algorithm will sharply grow. In such cases, our approximative approach should instead be used.

6.7 Related Work

Our work touches a number of different areas, so we briefly discuss related work in each area in different subsections. First, the recent work on quality analysis of SPLs is overviewed. Then we provide a short discussion on the approaches using UML for modeling SPLs. This is followed by a brief discussion of the recent work on model checking applied to SPLs.

6.7.1 Quality Analysis of SPLs

Analysis of quality attributes (QAs) of software systems has been extensively studied in the last years. However, most work has been done on analysis of single systems. Lincke et al. [59] highlight the importance of QA prediction models, and discuss how they impact on QA evaluation. Göbel et al. [36] present a component model *COMQUAD* to separate functional and non-functional aspects of component-based systems. The existing technologies, like Enterprise JavaBeans and CORBA Components, are used to describe non-functional aspects and to dynamically bind components to the running applications.

Different techniques have been proposed to evaluate the reliability of a software application. For example, Rodrigues et al. [74] present an approach to predicting reliability of component-based systems. The behavior of system is specified through scenarios in which components interact with each other. Message Sequence Charts are used to model scenario-based behaviors of components, which are then translated into probabilistic Labeled Transition Systems (LTSs). The resulting LTSs are synthesized and analyzed to calculate the system reliability. Immonen and Niemelä [48] provide a comprehensive survey of reliability prediction techniques existing in the literature. The techniques are reviewed and evaluated to see how they can be applied to software architecture. The authors conclude that none of the existing approaches address variability in software architecture. Accordingly, the literature lacks reliability prediction techniques for SPLs.

Despite the importance of quality analysis, there has been a limited work addressing this issue in the area of product line engineering [57]. Etxeberria et al [30, 57] propose a method, including a set of activities, to support and manage QAs in an SPL development lifecycle. The authors in [67], propose a framework to capture and elicit QAs of software product lines. Essentially they argue that different stakeholders of a product may expect different QAs, and this variability shall be identified and documented. Moreover, they describe how to map such varying requirement to software architecture. Jarzabek et al [49] integrate goal-modeling and feature-oriented modeling, and provide a notation feature-softgoal interdependency graph (F-SIG) to specify the interdependency between QAs and various features. However, the approach doesn't address variability modeling in system behavior, and is limited to feature models.

Zhang et al [91] study the prediction of QAs for SPLs by exploiting the application of Bayesian Belief Networks (BBNs). The QAs and features are enumerated and the impact of features on QAs is specified in terms of probabilistic values (by using BBNs). The overall quality of each configuration is calculated by solving the BBNs. In [6], Bartholdt et al. propose a methodology to evaluate quality of different products of an SPL. The underlying model used in the methodology is limited to feature models. QAs of each alternative are estimated (e.g. reliability of alternative A is 0.95), and the overall quality of each product is calculated by using a defined aggregation function. The aggregation function varies depending on the nature of QA, and in the simplest case is a summation. However, a challenging part of the approach is how to obtain the aggregation functions, and how to know if they are relevant and precise. Moreover, the approach only focuses on feature models, which are not suitable for evaluating a class of QAs like reliability, performance and energy consumption. The reason is that analysis of these properties is typically performed based on behavioral models [52].

An approach to estimate performance attributes of different products of an SPL is presented by [85]. The approach is applied to behavioral models specified in UML. The annotated UML diagrams are transformed into Linear Queueing Networks that can be analyzed by existing tools. However, the approach is based on analysis of each product independently, which can be very expensive in case of a large number of products. On the contrary, the authors of [29] focus on the importance of quality analysis of product lines and discuss how it can be more scalable through architecture evaluation as the common part of a software product line. In particular, as for performance analysis they suggest that execution graphs can be derived from code and mathematical formulae can be created by analyzing the graph. In the formulae, variable parts of the architecture are represented by variable performance properties. Having those formulae, it is very simple to evaluate each product by replacing the variables with real values. However, it is not discussed how to obtain those formulae in an automatic and precise way.

In the contrast to the model-based approaches, there exist some work on evaluation QAs of SPLs by using source code. Siegmund et al. [81,82] propose techniques to measure QAs of SPLs using the source codes. In [82], they focus on three QAs: *maintainability*, *binary size*, and *performance*. Furthermore, an optimization approach is proposed to select a product that satisfies a set of requested QAs. Since measuring all the possible configurations of large-scale SPLs is very expensive, an approximative approach is proposed in [81]. Similarly, Sincero et al. [83,84] present an approach and provide the tool support to configure products of an SPL based on QAs. Since testing QAs of all the products is expensive, a limited number of products are selected for the evaluation. The reason beyond such approach is that some features do not affect QAs. Features are labeled as selected, blocked, and open, depending on their impact on QAs. The selected features must exist in any product, and open features are optional. By excluding the blocked features in deriving products, the number of products is reduced and as a consequence the evaluation cost decreases. Obviously, the approaches based on available source code are not applicable in the early stage of development, when only models of the system exist.

6.7.2 UML for SPLs

There exist different papers on the application of UML for product families. Gomaa [38] provides guidelines to use UML in developing software product lines. He also describes how to use different UML diagrams such as Use Cases and Communication Diagrams to model SPLs. In [93], Ziadi et al. discuss the use of stereotyping techniques to extend UML to specify variability. The main advantage is that the current modelling tools can be used also for extensions. Cengarle et al. [12] discuss variability modeling in UML and provide mathematical models for that.

Since we are only using SDs in our approach, we only focus on representing variability in this kind of diagram. We follow a slightly changed version of the approach used by [85], and use the stereotypes <<variation point>> and <<optional point>>, which are shown by Alternative and Option combined fragments respectively. Moreover, variation points are linked to a feature model used to organize features and variants.

6.7.3 Model Checking of SPLs

Formal verification and model checking techniques provide a powerful tool to investigate the correctness of software systems. Recently, there has been an interest in proposing approaches for model checking of SPLs. Classen et al. [20] present an approach to verify all products of a product line in an efficient way in comparison to independent model checking of all products. The approach supports verification of LTL formulae. The result shows that the approach is able to reduce the model checking time up to 7 times. In [55], the authors propose an approach to verify CTL formulae against an SPL. They use variable I/O automata to specify product line behavior, provide an algorithm to verify CTL properties, and pinpoint the products which are not able to satisfy properties. The work we referred to mainly focuses on verification of functional properties. They do not support quantitative analysis of non-functional properties, for which probabilistic model checkers are specilized.

6.8 Conclusion

Through this chapter we described our approach to modeling stochastic software product lines using both high-level and mathematical notations.

Moreover, we presented different verification techniques to check non-functional properties of all products of an SPL. To achieve this aim, we presented new model-to-model transformation as well as model checking techniques.

CHAPTER 7

Achieving Non-Functional Requirements at Run time

"*Again, you can’t connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future. You have to trust in something - your gut, destiny, life, karma, whatever. This approach has never let me down, and it has made all the difference in my life.*" Steve Jobs

7.1 Introduction

In this chapter, we extend our approach of model-checking SPLs to build adaptive systems that can continuously check the satisfaction of NFRs and achieve them by planning and applying suitable adaptations. The proposed solution is based on a holistic approach that covers both design and the run-time phases of the application lifetime. A design-time verification phase is integrated with continuous run-time verification and reconfiguration that support the adaptation. This is similar to *Incomplete Model Checking* approach described earlier, but targets NFRs, namely reliability and energy consumption.

First, at design time, software application is designed as a *dynamic software product line* (DSPL). An SPL defines a *family* of software products that can be viewed as *configurations*. In our case, configurations differ in the way they satisfy the same requirements in different contexts. The different contexts are represented during design as *variation points* in product-line terms. The product line, in our approach, is dynamic in the sense that the different instances are generated dynamically at run time. Moreover, our notion of a "product" is unconventional, in the sense that it does not refer to code, but to higher-level models. The models we refer to are the ones that are used by software engineers during design to reason about the NFRs of interest for applications.

All our reasoning and manipulation is performed at the model level. In particular, models are also used to describe configurations. The way models are transformed into implementations and then into deployed units is ignored here. We simply assume that this can be done by following some systematic model-driven development strategy. We focus instead on how design models and specify variation points and variants, and finally how to check NFRs and plan adaptations at run time. The method we propose starts from a high-level specification of the system and of the environment, given in terms of SVSDs.

At design time, SVSDs –that describe the entire (model level) product line– are transformed into parametric DTMCs, and then verified against NFRs. Environment conditions are in fact also described as part of the models. The main contribution is to show how to avoid separate verification of each configuration through a novel approach that exploits commonalities among different configurations, which are factored out to support efficient verification. At run time, whenever the (model of the) currently running instance of the application is found to violate the requirements, because of environmental changes, another instance is identified (and the corresponding target implementation is deployed) that does satisfy the requirements under the new external conditions.

We continue this chapter with a running example, which motivates the type of problems we address. Then we briefly overview the advanced techniques in the literature, and then present our approach.

7.2 Running Example

In the following, a running example is described for which we aim at building an adaptive system.

"The Happy Hour Organizer (HHO) is a system to help people socialize as they move around in a modern city. The system is developed in order to make organization of daily social events easy and as automatic as possible. One of the scenarios that this system supports is about "grouping" in impromptu meetings. To achieve that, the system helps people, who have the same interests, to find each other and perform a social activity (which we call Happy Hour). For instance, someone may like to meet other people who study the same foreign language to practice in conversation. Thus they may have a nice evening in a bar while sharing their knowledge about the language. To organize such impromptu meetings, the HHO application running on the user's smartphone looks into a social network and searches for other people around city and especially near her place. The system obtains the user's current position, and takes it into account while selecting and contacting people. The application finds a group of people and communicates with their devices to make an agreement on the appointment time. Later, the application searches and books a place like a bar or pub in which the event can be held.

The system is to satisfy two NFRs concerning reliability and energy consumption. More precisely, the whole scenario shall be performed with a reliability higher that 0.95 and maximum energy consumption of 1000."

The running example described above includes variation points both in the system and environment. For example, the mobile system needs to detect the current position of the user through a *locator* device. This functionality can be performed via two embedded components: GPS or GSM. Another example is the communication service between different devices, which can be performed either via WiFi or SMS. These two are examples of internal variabilities developed as a part of system. Examples of the variation points in the environment are the *social network* and the *place booking* services. Many external applications exist to support these services, and their invocation corresponds to external variation points, whose variants can be found in the environment. Different variants may be visible or not depending on the physical location; they may appear and disappear over time; they may provide low or high-level QoS. Therefore, it is important for the system to switch between the variants which can better fit HHO's functional and non-functional requirements. Indeed, the main challenges are how to select a configuration and how to make sure that it continuously satisfies the non-functional properties. One issue is that different non-functional requirements may have a conflicting nature. For example, a reliable component may consume more energy than an unreliable one. Therefore, finding a set of variants that altogether provide a good quality with respect to the requested properties can be a difficult task.

7.3 Self-Adaptive Systems for NFRs Satisfaction

The Rainbow framework described by Garlan et al. [14] represents one of the earliest attempts to support self-adaptation of software systems. Adaptations are prescribed as script rules for different foreseen problems at design time. Calinescu and Kwiatkowska [11] introduce a framework to implement autonomic systems in order to optimize satisfaction of NFRs. The framework mainly relies on policies specified by users, which are defined over configurable parameters. Adaptation planning is performed by exhaustively searching for optimal values of configurable parameters. Maximum size of a queue and connection database pool are two examples of configurable parameters. This approach does not support architectural adaptation. This can be a shortcoming because most of modern systems are comprised of black box components and only architectural adaptations are possible [64]. Adding and replacing components are two examples of architectural adaptations. Also the approach applies classical model checking against all possible configurations. This may result in inefficiencies due to large number of possible configurations.

To make adaptation planning flexible, Kim et al. [51] and Elkhodary et al [26] exploit the use of learning algorithms. In particular, Kim et al [51] investigate the use of *reinforcement learning* techniques to enact dynamic adaptation plans at run time. They propose an approach to Q-Learning-based action planning in which in any given situation an appropriate adaptation is selected. After performing an adaptation, the system receives a reward that represents the effectiveness of the applied adaptation. The reward is used to tune the parameters of the learning functions which select the next adaptations. The main problem of the learning-based approaches is the learning period that the algorithms require for tuning parameters.

The above approaches and their characteristics are summarized in Table 7.1. An approach like the one pioneered by [14] uses design-time incomplete knowledge to provide adaptation rules, while others like [51] [26] are based on run-time learning. Both of them may lead to failure in adaptation planning. Prescribing adaptation plans at design time can be very risky due to incomplete knowledge. On the other hand, applying only run-time approaches needs a long learning time after system deployment. Our aim is to reach a balance between design time and run time. We embed adaptation points into system models and make sure that possible configurations can satisfy NFRs with respect to design-time assumptions. If design-time assumptions are not violated, then the running system is guaranteed to satisfy the requirements. However, in case the assumptions are violated, we apply light-weight run-time planning techniques. To be more clear, at run time we collect environmental data, update models, and apply evolutionary techniques to find candidate configurations. We ensure that a selected reconfiguration improves NFRs satisfaction. Our verification-based run-time approach is efficient because we use parametric verification techniques that are computationally expensive only at design time. Thus, the run-time overhead of planning is minimized.

Table 7.1: *Model-based approaches to build adaptive systems*

Approach	Specification	Planning	Adaptation	Run-time Overhead
Garlan et al [14]	static	design time	architectural	rule reasoning
Calinescu et al [11]	behavioral	run time	parametric	exhaustive search
Kim et al [51]	static	run time	architectural	learning

Another distinctive feature of our approach is the use of high-level behavioral models for system modeling and NFRs analysis. This is important specially due to the nature of NFRs like reliability and cost, which are highly dependent on system behaviors. For example, the number of repetitions of an activity can have an impact on reliability of a system scenario. Behavioral models can precisely predict future satisfaction of NFRs. Calinescu et al. [11] also base their approach on the use of formal models of NFRs, but they do not focus on architectural adaptations. To achieve this goal, we exploit the application of DSPLs in the design of adaptive system.

7.4 Dynamic Software Product Lines

DSPL is a new type of SPL [21] in which variation points may be frequently rebound at runtime [41]. While SPL aims at improving productivity and reducing the time and cost for developing a family of products, DSPL is a way to build self-adaptive software. The main difference between an SPL and a DSPL is the binding time of variation points to variants. In an SPL the binding is established *before* run time, while in a DSPL the binding is established later at run time. Indeed, while SPLs are used to deal with variability of the market, DSPLs are applied to cope with changing environments and individual requirements.

According to Cetina et al. [13], "DSPLs encompass systems that are capable of modifying their own configuration with respect to changes in their operating environment by using run-time reconfigurations". According to Hallsteinsen et al. [42] [41], DSPLs support dynamic variability, frequent run-time binding, and user requirements change. Moreover, they may provide the capabilities for context-awareness, self-adaptation, and autonomic decision making. In [56], context-awareness, resource-aware decision making, permanent service delivery, and consistent dynamic reconfiguration are considered as the major properties of DSPLs.

There exist a few research efforts aiming at modeling and developing DSPLs. Morin et al. [66] describe how DSPLs may be architected to manage dynamic adaptations. Trinidad et al. [87] model a DSPL by using feature models. Bencomo et al. [7] discuss how to capture and model variability of adaptive systems through SPL modeling approaches. They use feature models to provide a structural view of system variability, and apply transition diagrams to specify system reconfigurations in response to environmental changes. In [60, 69], the authors discuss the application of aspect-oriented methods in developing DSPLs.

Lee and Kang [56] point out the importance of DSPL to address unexpected changes in environment and focus on dynamic reconfiguration as important means. They describe how to use feature models to represent variation points and how to switch between different configurations with respect to system context. Our approach follows a similar path, although our main focus is on using a DSPL-based software architecture to achieve run-time adaptations that enable continuous satisfaction of NFRs. Additionally our approach relies on efficient formal analysis of NFRs, which dynamically supports architectural adaptation planning.

7.5 The Proposed Approach

The overall view of the approach is shown in Figure 7.1. As illustrated, the framework covers both design time and run time. During design time, the aim is to design a DSPL, specify architectural models, and analyze them against expected NFRs. At run time, while the DSPL starts operating in environment it keeps monitoring quality data that may affect NFRs satisfaction. The requirements are continuously verified with respect to run time data that may reflect changes in the environment's behavior. In the case of detection of any violations, adaptation plans are generated and applied. In

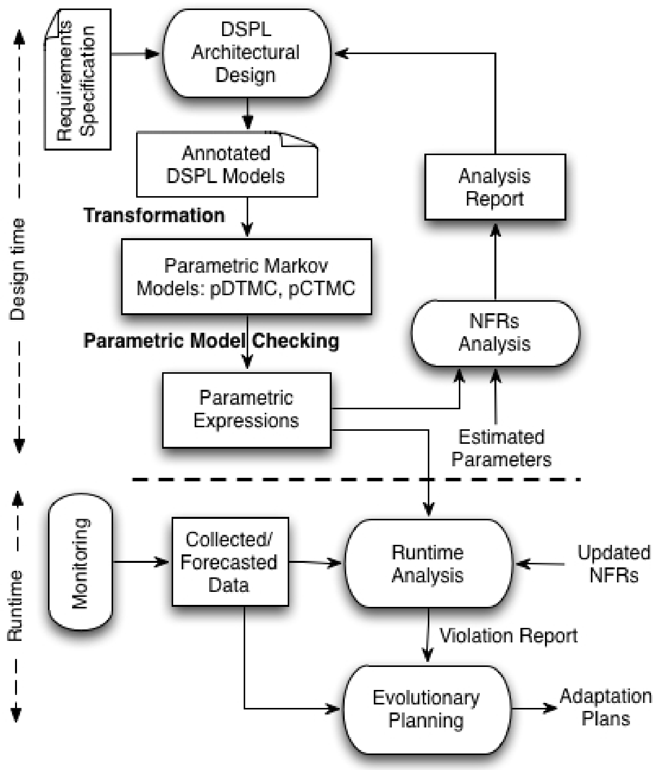


Figure 7.1: *The Proposed Framework*

the following, we discuss each phase in turn and describe the relevant activities.

7.5.1 Design Time

The framework starts at design time when the architecture of the DSPL is designed through a feedback loop. The key point of design is to introduce variation points through which adaptations can be performed. The architectural design is then verified against expected NFRs by using *parametric model checking*. As we will see below, the different configurations—resulting from different instantiations of variants—are model checked in the different environment conditions for which they are conceived. The goal is to show whether or not the different configurations can satisfy NFRs. The designer can check the analysis results and may modify the architecture accordingly. In the sequel, we briefly discuss the techniques used in design-time activities.

Modeling

The main issue of modeling a DSPL is to specify variation points and variants. The system is designed as usual but the adaptive parts are specified as features, for which there exist alternative choices. The abstract feature model of the running example is shown in Figure 7.2. Every feature is a functionality that may be achieved using different variants. Variants can be implemented as a part of the system or may be hired as external services in environment.

In our framework, the behaviors of a system are specified by using Sequence Diagrams (SDs). Furthermore, new stereotypes (<<variation point>> and <<variable>>) are added to represent varying behaviors. The former (see Figure 7.3(a)) describes the choice between variants in the system's architecture (*internal variability*). The variation points are represented by fragments combined through the alternative (labelled *alt* and *else*) and stereotyped as <<variation point>>. External services whose selection may be performed at run time to achieve dynamic binding are modeled as an invocation from a component, stereotyped as <<variable>>. Figure 7.3 shows the two kinds of varying behaviors and Figure 7.4 illustrates the SD for the running example. This represents an *external variability*. Concerning external services, every variation point is modeled as an invocation of a service from an abstract component, which is discovered in environment at run time. *Social Network* and *Place Booker* are two examples of external services, while *Location* is an internal variation point in the running example, referring to GPS and GSM as variants.

Similarly to the previous chapter, SDs can be annotated with quality data by following the UML MARTE profile. In particular, each message is annotated with two tags: *prob* and *energy*. The former represents the probability that a message is successfully transmitted; the latter expresses the amount of energy consumed to transmit a message.

Model-to-Model Transformation and Parametric Verification

Our goal is to ensure that NFRs are satisfied by the system while it is executed. One possibility would be to use traditional model checking to achieve this goal. In this case, at design time we would model check all configurations in the different environment conditions in which they are supposed to work. Whenever at run time the current configuration is executed, its model would also be analyzed by the model checker in the current

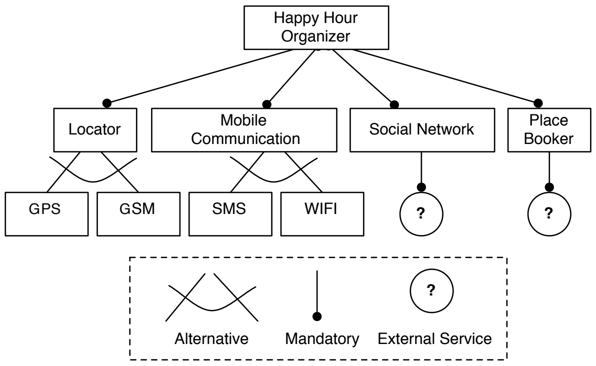
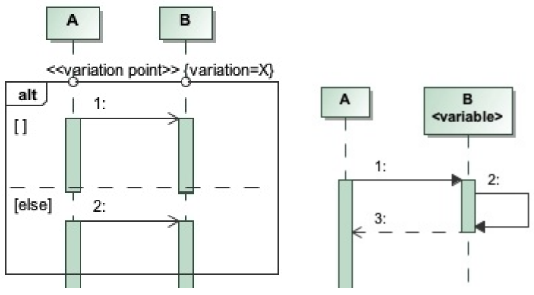


Figure 7.2: *The feature model for the running example*



(a) Variation Point (b) Variable

Figure 7.3: *Varying behaviors*

environment conditions. A failure of the model checker to satisfy the requirements would then drive the selection of an alternative configuration. This approach, unfortunately, is unlikely to work in practice, especially because of the time required by the analysis step, which may lead to unacceptably late reactions. This is where parametric model checking comes into play. To make run-time verification feasible, we apply a parametric verification approach instead of the classical one. In this case, parametric verification is performed at design time and a formula is generated, which is later evaluated quite efficiently at run time when updated real data are available.

Our approach is intuitively shown in Figure 7.5. For the sake of simplicity of presentation, we assume here that variants themselves do not contain any variation point. Note that handling nested variation points requires a simple hierarchical process, and does not impose extra effort. As illus-

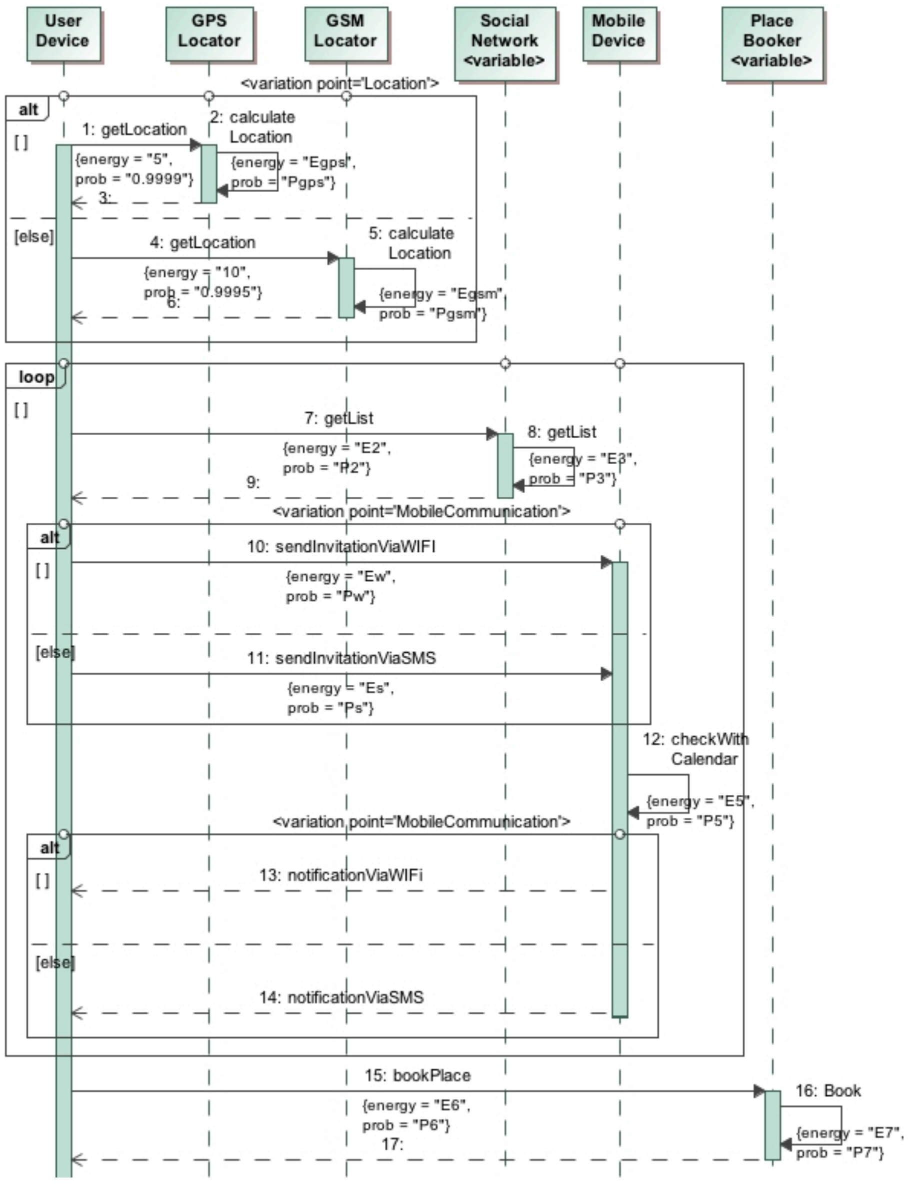

User Device
GPS Locator
GSM Locator
Social Network <variable>
Mobile Device
Place Booker <variable>
<variation point='Location'>
alt
[ ]
1: getLocation
{energy = "5", prob = "0.9999"}
2: calculate Location
{energy = "Egps", prob = "Pgps"}
3:
[else]
4: getLocation
{energy = "10", prob = "0.9995"}
5: calculate Location
{energy = "Egsm", prob = "Pgsm"}
6:
loop
[ ]
7: getList
{energy = "E2", prob = "P2"}
8: getList
{energy = "E3", prob = "P3"}
9:
<variation point='MobileCommunication'>
alt
[ ]
10: sendInvitationViaWIFI
{energy = "Ew", prob = "Pw"}
[else]
11: sendInvitationViaSMS
{energy = "Es", prob = "Ps"}
12: checkWith Calendar
{energy = "E5", prob = "P5"}
<variation point='MobileCommunication'>
alt
[ ]
13: notificationViaWIFi
[else]
14: notificationViaSMS
15: bookPlace
{energy = "E6", prob = "P6"}
16: Book
{energy = "E7", prob = "P7"}
17:

Figure 7.4: *The annotated SD for the running example*

trated, a DSPL model—an annotated SD—is divided into a core part and its variants. Through such process, variation points are transformed into messages annotated with variables. If a variation point represents an internal variability, it is transformed into a self-message annotated with two variables $P\#$ and $E\#$ standing for *prob* and *energy* respectively, meaning that their values depend on the alternative that is chosen in the configuration. The values for these variables may be computed by model checking each alternative, treated as an independent behavior. Each alternative thus goes through a similar verification process, since in general (but not in the simplified case assumed in Figure 7.5) it can contain further variation points and variants.

Figure 7.6 shows the parametric SD generated for the running example. As shown, GetLocation is an example of the self-message replaced for the variation point Location. In the case of external variation points (external services), their quality annotations are represented as variables $P\#$ and $E\#$. For instance, this applies to the place booking service of our running example. Variables are also used to label transitions that correspond to environment phenomena that may change at run time. For example, in some other interactive applications we may lack information about user preferences, such as the probability that one of two alternative options may be chosen by users and may affect the way requirements may be satisfied. To model this situation, it is possible to introduce an alternative in the SD and labeling each option with a (variable) probability [1]. In conclusion, this design-time step leads to the derivation of a *parametric SD*, where variables instead of constant values are used as annotation tags.

To evaluate NFRs, parametric SDs are transformed into parametric Markov models, similarly to what explained. The transformations from SDs into Markov models are performed by following the approach described in the previous chapter. Regarding Markov models, parametric DTMCs are used to verify reliability properties, while parametric DTMCs with Rewards are used to verify cost properties (typically, energy, CPU, or network usage). NFRs are expressed as formulae written in formal languages PCTL or as Cost/Reward properties. Figure 7.7(a) represents the parametric DTMC corresponding to the SD in Figure 7.6, which can be used to reason about reliability concerns. Examples of non-functional properties we would like to state are expressed as below. Note that state 13 of the parametric DTMC (Figure 7.7(a)) is the state that corresponds to the condition PlaceBooked mentioned in the properties.

[1]There must be an additional constraint that their sum equals 1

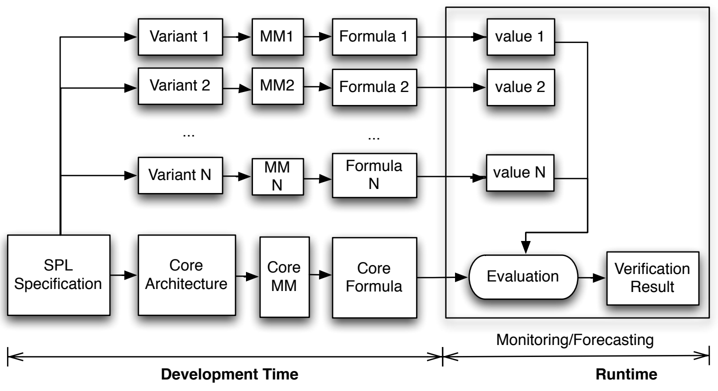


Figure 7.5: *Parametric Verification of SPLs - MM stands for Markov Model*

$$P => 0.95[F(State = PlaceBooked)] \tag{7.1}$$

$$R =< 1000[F(State = PlaceBooked)] \tag{7.2}$$

The first property states that the probability of reaching a state in which the meeting place is successfully booked shall be greater or equal to 0.95. Note that this is the final state of the whole scenario, so the property expresses a constraint on the probability of having whole scenario successfully completed. Similarly, the second property states that the whole energy consumption shall be less or equal to 1000.

To evaluate requirements satisfaction, the parametric Markov models and the property formulae are fed into a parametric model checker. The resulting formulae of the verification for the reliability and energy properties of the running example are presented below. These formulae are used for two purposes. First they are used for design time verification of different configurations. In this case, we have to make assumptions about quality data for the parameters. The values we select represent the environment conditions we predict as possible, and for which we want to prove that an appropriate configuration exists that can satisfy the NFRs. In case no configuration is able to satisfy the NFRs, the designer should change the DSPL architecture. Furthermore, these formulae are used for run-time analysis and planning to perform continuous verification and self-adaptation.

$$reliability = \frac{P6^2 * P1 * LP * P7 - P6^2 * P1 * P7}{PL * P3 * P2^2 * P5 * P4^2 - 1} \tag{7.3}$$

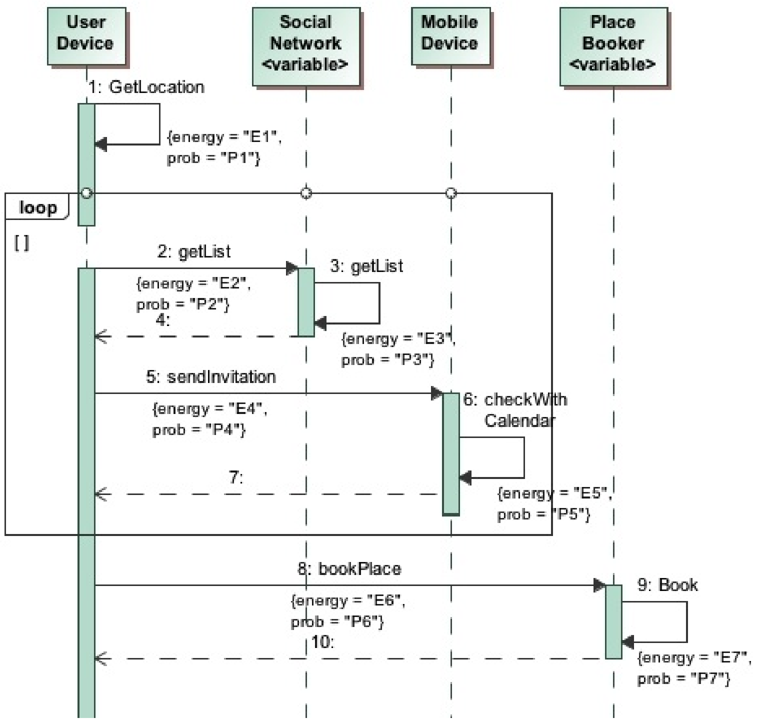

User Device
Social Network <variable>
Mobile Device
Place Booker <variable>
1: GetLocation
{energy = "E1", prob = "P1"}
loop
[ ]
2: getList
{energy = "E2", prob = "P2"}
3: getList
{energy = "E3", prob = "P3"}
4:
5: sendInvitation
{energy = "E4", prob = "P4"}
6: checkWith Calendar
{energy = "E5", prob = "P5"}
7:
8: bookPlace
{energy = "E6", prob = "P6"}
9: Book
{energy = "E7", prob = "P7"}
10:

Figure 7.6: *The proposed framework at run time*

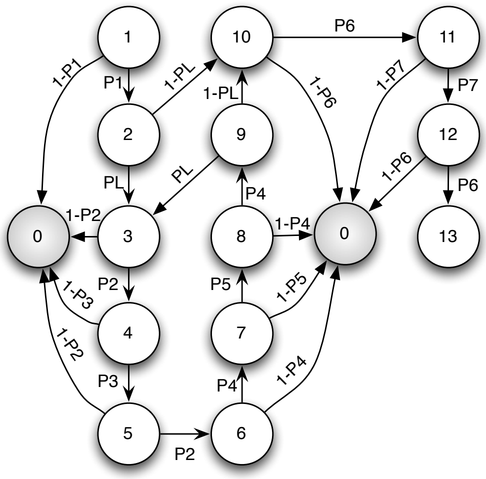
$$energy = \frac{2*E6*PL-2*E6+1*E1*PL-E1-PL*E3-2*PL*E2-PL*E5-2*PL*E4+PL*E7-E7}{PL-1}$$

As mentioned, each variant is also transformed into parametric Markov models. Figure 7.7(b) shows the parametric DTMC corresponding to the selection of the GPS locator. For each variant, reachability properties are in turn evaluates on the respective Markov models. Note that in fact every variant has a behavior that starts from a starting state and ends in one or more final states. The properties are shown in the formulae below.

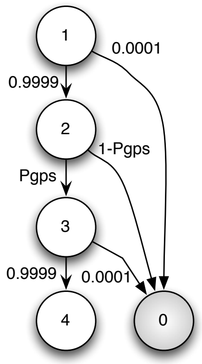
$$P =?[F(State = End)] \tag{7.4}$$

$$R =?[F(State = End)] \tag{7.5}$$

The verification of Markov models against every property also results in a formula. After providing quality data for the parameters, the formula is evaluated by substituting real numbers. The real number is the quality (reliability or energy) that a variant can provide. To evaluate the quality of



(a) Parametric DTMC for the core behavior of the running example - state '0' is drawn two times to make the figure readable.



(b) Parametric DTMC for the alternative behavior of using GPS

Figure 7.7: *Parametric DTMCs*

the whole scenario, the quality of variants are fed into the parameters of the main formula.

7.5.2 Run time

When the framework moves to run time, its activities are inspired by MAPE-K cycle shown in Figure 7.8, popularized by the autonomic computing research community (see [50]). The quality data collected through monitoring must be transformed into values that can be used to feed the formula. This transformation in general depends on the abstraction that model parameters realize on the physical data measurable in the environment. As a typical example, physical data may represent the detected failures of external service invocations, whereas model parameters may represent service reliability expressed as a failure probability. In general, the transformation process from environmental data collected by monitors to model parameters can be quite complex and may require approaches based on machine learning. An example is presented in [28].

Hereafter we assume that suitable transformations from monitored data to model parameters exist in the run-time environment. Updated parameters are used to evaluate the parametric formulae in order to analyze the current satisfaction and also to foresee future NFR violations.

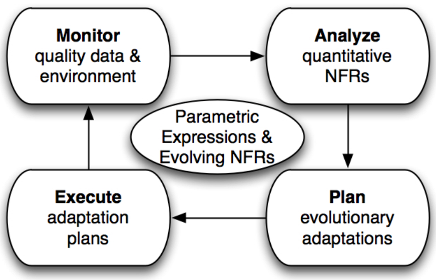


Figure 7.8: *The proposed framework at run time.*

As for the knowledge base, parametric formulae and NFR properties are kept at run time for analysis and planning purposes. In case the analysis detects or predicts any violations, planning techniques are used to generate adaptation plans by which the system can optimize its behaviors. For this purpose, we employ evolutionary algorithms and in particular Hill Climbing (HC), which is able to find a solution that represents a good trade-off between precision of the results and timeliness of the provided response. As a result, adaptation plans are generated and applied as a new configuration from the DSPL. An architectural adaptation can therefore be simply seen as set of variant substitutions for given variation points. However, the main issue of planning is to find a configuration of variants that optimizes the satisfaction of possibly conflicting properties. In general, there might be various NFR properties (e.g. performance and energy consumption) that may have competing nature in the way they can be taken care of in an implementation. For example, regarding a variation point X, there may be different variants providing the same functionalities but different quality properties. Variant A may provide a high response time with a low energy consumption, variant B may provide a low response time and a high energy consumption, and finally variant C may provide an average quality for both cases. Due to the existence of different variation points, finding a configuration that optimally satisfies most of the NFRs can be challenging.

If the verification of the current DSPL configuration fails at run time after updated quality parameters are fed into the verification formulae, a reconfiguration plan is activated to perform a chain of adaptations. In terms of DSPL, an adaptation is a substitution of a variant with another one that would help the system to better achieve its requirements. However, exploring all possible combinations of variants needs exponential time, and is inherently an NP-Complete search problem. It is true that by using parametric model checking and avoiding the whole run-time model checking process, the required time is reduced, but in case of a large number of variation points and variants, evaluating all combinations can be impossible at run time within the time limits within which a reaction to NFR violations must be enacted. Therefore, we apply an evolutionary approach, like HC, which takes into account the constraint on the available reconfiguration time and finds a sub-optimal configuration. The algorithm is able to provide a more accurate solution (i.e., one that is closer to the optimum) if more time is allocated to the search. The HC algorithm, in general, searches to find a sub-optimal solution considering a budgeted time. The search continues until a solution is reached, which represents a good candidate to be chosen for adaptation, given the limited time available to perform the search.

HC is an optimization method that iteratively searches for better solutions. It starts with a random solution, then tries to improve it by iteratively changing a single element of the solution. If the change leads to a better solution, the change is applied. The process is continued until new improvements cannot be found. HC does not guarantee that the resulting solution is the best possible solution. However, it can find a better solution than other algorithms when the available search time is limited. The remainder of this section provides an intuitive, high-level description of how we apply HC to generate a new configuration of a DSPL.

Let us consider $P = \{p_1, p_2, ..., p_N\}$ be the satisfaction degree of the set of NFR properties that a DSPL is supposed to satisfy. The elements of the set represents the degree that a given configuration satisfies the properties. For each property, a weight number is introduced that expresses the importance of the property. The weights are expressed as a set of real numbers $W = \{w_1, w_2, ..., w_N\}$. Therefore, the *total utility* of a selected configuration can be specified as:

$$U_C = w_1 * p_1 + w_2 * p_2 + ... + w_N * p_N \tag{7.6}$$

Regarding our running example, there are two properties (p_1, p_2) corresponding to reliability and energy consumption, respectively. We consider $(w_1 = 2, w_2 = 1)$ as the weights for those properties, which means that the importance of reliability is "twice" the importance of energy consumption. Note that the energy consumption property is normalized by dividing the currently measured value by the maximum energy consumption expressed in the requirement.

Algorithm 1 shows the pseudocode for the HC approach. The algorithm starts with the current configuration of DSPL. It iteratively searches for any other configuration that can better satisfy the properties. To do that,

Table 7.2: *Reliability and energy parameters of the running example.*

Reliability	P1	P2	P3	P4	P5	P6	P7
-	0.9	0.998	0.995	0.95	0.998	0.995	0.998
Energy	E1	E2	E3	E4	E5	E6	E7
-	50	60	70	40	50	40	30

the algorithm randomly chooses a variation point and replaces one of its variants. Then, the total utility of the new configuration is calculated. If it is greater than the utility of the current configuration, it is selected as a candidate configuration. The algorithm continues to randomly search for other candidates that are better than the new selected configuration. This procedure is carried out until the limited time is finished. In the end, the difference between the initial configuration and the selected configuration is calculated in terms of variant substitutions.

Algorithm 1 Hill Climbing Algorithm

```
**HillClimbing** ($VarationPoint[]\ VP, Variant[]\ VR, Configuration\ cf$)
{
 **var** $tempCf\ =\ cf$;
 **while**($timeLimit < 0$)
 {
  $vnt\ =\ ChooseVariant(VP, VR)$;
  $newCf\ =\ Combine(tempCf, vnt)$;
  **if**($Utility(newCf) > Utility(tempCf)$)
  $tempCf\ =\ newCf$;
 }
 **return** $Diff(tempCf,\ cf)$;
}
```

Let us consider as an example how the proposed approach works for our running example. Of course, the example does not show the real value of HC, which would become evident only in the case of a very high number of alternatives to evaluate. Assume that (see Figure 7.9) the currently running configuration for the HHO application uses GPS and SMS as the internal variants. Also assume that the user moves around and changes her physical context. It may then happen that the quality parameters change and the NFR properties (1) and (2) are not satisfied any more. The updated quality parameters are shown in Table 7.2.

A violation is discovered by evaluating the parametric formulae (3) and (4) for both reliability and energy consumption considering the updated

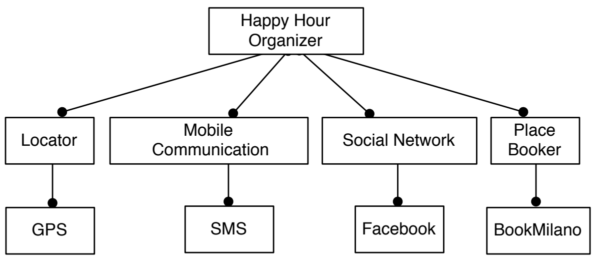


Figure 7.9: *The violated configuration using GPS and SMS.*

Table 7.3: *New reliability and energy data for the quality parameters.*

Reliability	P1	P2	P3	P4	P5	P6	P7
-	0.995	0.998	0.995	0.999	0.998	0.995	0.998
Energy	E1	E2	E3	E4	E5	E6	E7
-	55	60	70	45	50	40	30

parameter values. In fact, the evaluation results in 0.73 for the reliability property (1), which is much less than 0.95 as the expected minimum. To deal with such violation, the HC algorithm is applied and a configuration using GSM and WiFi is selected as the new configuration (Figure 7.10). As the result, the application shall apply two adaptations: substituting GSM for GPS, and WiFi for SMS. Using this configuration, the reliability and energy consumption properties are evaluated to 0.95 and 836, which satisfy both NFR properties. The updated parameters of the new configuration are shown in Table 7.3.

The application keeps monitoring and updating the quality parameters,

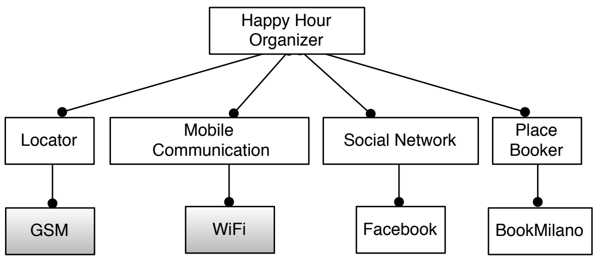


Figure 7.10: *The new configuration after applying the adaptations.*

and feeds them into the parametric formulae in order to discover future violations. For the sake of simplicity, we did not discuss the QoS changes of external services in this example. In the example, Facebook and BookMilano are used as the external services. Similarly to internal variabilities, it can be the case that their QoS changes which may lead to property violations and further adaptation planning.

7.6 Conclusion

DSPLs can be viewed as a clear approach to capture and specify variability and further more adaptations. Our approach can be extended for other specification languages and requirements. It is built on top of existing techniques and tools, and can be completed beside a model-driven development to more automatize both design-development and run-time adaptation.

Part IV

Conclusion

CHAPTER 8

Conclusion and Future Work

"*There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened.*" Douglas Adams

In this chapter, we conclude our journey through verification techniques for Variability-Intensive Systems (VISs). We recall the research questions introduced in Chapter 1, and describe how they are addressed through this thesis. Then we discuss the current limitations and future perspectives.

8.1 Reviewing Research Questions

Let us review the research questions, and discuss them one by one.

RQ.1. *Given an incomplete system specification, how to check whether or not the system behavior satisfies the expected requirements?*

An innovative technique – *Incomplete Model Checking* (IMC) – to deal with kind specifications is presented in Chapter 4. It is later extended by an incremental methodology - AGAVE - to verify requirements as the system is evolved and gradually developed 5. By the time, this approach is investigated for state-based behavior specifications against CTL-like properties, so we leave studying other temporal logic languages as the future work.

RQ.1.1. *How to formally specify an incomplete system behavior?*

The notion of incompleteness in this thesis is essentially focused on unspecified parts of a system behavioral specification. These parts are encapsulated and viewed as a functionality, which may be only a single action or a complicated sequence of actions. In Chapters 4 and 5, two state-based formalisms LTS and Statecharts are enriched with *Transparent* states, which represent unknown behaviors. In the former case, ILTS provides a compositional formalism in which transparent states are subject to be refined into further transition systems. Similarly Statecharts can evolve through transparent states, each of which is refined into another Statechart.

RQ.1.2. *What language can fully specify the set of constraints that lead to the satisfaction of a temporal property?*

Next-Free Path-CTL is the language we introduce to express constraints generated by a model checker checking CTL-like properties of an ILTS. The language adds a fresh temporal operator to specify *global* properties over finite paths. This question is important to answer for any other temporal logic language. For example, if we take LTL as the property specification language, then the language to specify constraints shall be studied.

RQ.1.3. *How can we generate constraints over unspecified elements of an incomplete specification to guarantee that the original property holds?*

Chapter 4 presents the algorithm able to check ILTS against Next-Free CTL properties. Given a global property, constraints are computed to guarantee the property satisfaction. The algorithm is an extension of the standard CTL model checking [4], such that it is able to handle transparent states.

RQ.1.4. *As the system specification evolves, how can we re-use the results of previous verifications to calculate the new results?*

IMC is used as the core technique for incremental refinement and verification of Statecharts in Chapter 5. We show that the constraints generated for transparent states are checked later when the missing parts of specification is provided. It means that the whole verification is not performed whenever a piece of information is added. Instead, the constraints of the recently refined transparent state are checked. This approach allows us to avoid re-doing the verification for the parts processed earlier. Consequently, the satisfaction of the constraints will imply the satisfaction of the global property.

RQ.2. *How to specify and model-check stochastic behaviors of SPLs against non-functional requirements?*

In the third part of the thesis, we focused on Non-Functional Requirements (NFRs) and studied the techniques to capture stochastic behaviors of many products driven from an SPL, and their verifications against NFRs. Due to the variety of NFRs, we address the techniques only for two of them (namely *reliability* and *energy consumption*), which are of significance in practice.

RQ.2.1. *What are the suitable (high-level and low-level) formalisms to capture the stochastic behaviors of SPLs?*

We studied this problem in both high-level and low-level modeling languages. There exists a variety of approaches to introduce variability into high-level behavioral models, particularly UML diagrams [12, 38, 80, 85, 93]. In Chapter 6, we presented Stochastic Variable Sequence Diagrams (SVSDs) coupled with Feature Diagrams to capture both variability and stochastic information of an SPL. Obviously, different diagrams can be extended in this manner to offer both variability and stochasticity as native elements.

As for the low-level formalism, Featured DTMCs (FDTMCs) are proposed in which the notion of *feature* are added to DTMCs to differentiate the behaviors of different products. We show how to map SVSDs onto FDTMCs in order to further analysis.

RQ.2.2. *How to efficiently verify non-functional requirements of an SPL?*

Three different approaches (namely enumerative, parametric, and approximative) are proposed to verify an SPL – in terms of FDTMCs–, and to check all its products against reliability and energy consumption properties.

8.2 Limitations and Future Work

Similarly to every scientific work, the approaches described through this thesis may suffer from a number of threats and weaknesses. We discuss some of them in this section, and define the future work as well.

Regarding incomplete model checking, the approach is still in the preliminary stages, though it has been applied to a number of case studies and enjoys a tool support. A computational complexity analysis of the approach should be performed to complement the empirical evaluation. We consider IMC and AGAVE as young research fruits that can contribute to the literature of software engineering and formal verification. There are many directions to extend this work. At the moment, we are working to optimize the implementation and to explore a new symbolic approach. Regarding the formalisms, the future work is to support the full CTL by adding Next operator. Further steps are applying the approach to other case studies in different areas and extending the framework to support other temporal logics such as LTL. We are also working on relaxing the current restrictions of AGAVE such as side-effect free elaboration. Beside that we investigate how to mitigate the state-explosion risk of concurrent Statecharts.

Regarding stochastic SPL analysis, there are threats that challenge our model-based approach. First, high-level behavioral models are rarely available for SPLs. Due to this lack, we did not succeed to apply and evaluate our approaches w.r.t. industrial cases. Providing accurate stochastic information (e.g. failure probability of a server) for SPL behaviors at design time is another challenge, that constraints the applicability. However, we hope to take further steps by releasing an open source tool. In this regard, we aim at developing stochastic modeling languages for SPLs, as well as extending the approach to other non-functional properties, in particular to performance.

Dealing with open and closed variability together in a single specification could be another direction for future research. In this case, an SPL would contain both unknown components as well as variation points and alternatives. This could be specially interesting for adaptive systems in which both variabilities are subject to exist. Model checking such specification would result in more complicated constraints for unknown components of each valid configuration. Obviously handling such computation would be quite expensive for large-scale models.

Despite of our attempt to develop new verification techniques to address variability for different kinds of specifications, there exist many gaps that require further research. Most advanced model checkers do not support incremental evolution, so the whole verification is re-performed if there are tiny changes in system specification. This is an obstacle to integrate model checking with daily development environments, due to the fact that specification is mostly developed through the interaction with programmers and users and many versions are created, verified and debugged until obtaining a desired specification. Bringing incrementally and reusability to other verification techniques such as testing and static analysis of program code would be definitely of industrial interest.

Considering industrial applicability, the main threat is the lack of software models. Although there have been an extensive academic work on model-driven development, the realization of these techniques still seems a dream for software developers. While concrete programs and codebase are the main documentation of current software, extracting analyzable models and reasoning about them is a big challenge. Further research is required to bridge the gap between formal method experts and industrial programmers. The fact that formal verification requires highly skilled programmers is a reality that limits its practical usages.

Bibliography

[1] Rasmus Adler, Ina Schaefer, Tobias Schuele, and Eric Vecchié. From model-based design to formal verification of adaptive embedded systems. ICFEM'07, pages 76–95, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, May 2001.

[3] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, July 2009. (column).

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[5] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, July 2011.

[6] Joerg Bartholdt, Marcel Medak, and Roy Oberhauser. Integrating quality modeling with feature modeling in software product lines. In *ICSEA*, pages 365–370, 2009.

[7] Nelly Bencomo, Peter Sawyer, Gordon S. Blair, and Paul Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *Workshop on Dynamic Software Product Lines*, pages 23–32, 2008.

[8] Domenico Bianculli, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Interface decomposition for service compositions. In *ICSE'11*, pages 501–510, 2011.

[9] Friedemann Bitsch. Safety patterns - the key to formal specification of safety requirements. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*, SAFECOMP '01, pages 176–189, London, UK, UK, 2001. Springer-Verlag.

[10] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

[11] Radu Calinescu and Marta Kwiatkowska. Using quantitative analysis to implement autonomic it systems. In *Proceedings of the 31st International Conference on Software Engineering*, pages 100–110, 2009.

[12] Maria Victoria Cengarle, Peter Graubmann, and Stefan Wagner. Semantics of uml 2.0 interactions with variabilities. *Electronic Notes in Theoretical Computer Science*, 160:141–155, 2006.

[13] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Designing and prototyping dynamic software product lines: techniques and guidelines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 331–345, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, pages 276–277, 2004.

[15] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella.
Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[17] E.M. Clarke and W. Heinle. Modular translation of statecharts to smv. In *Technical Report CMU-CS-00-XXX Carnegie Mellon University School of Computer Science*, 2000.

[18] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking software product lines with snip. *International Journal on Software Tools for Technology Transfer*, 14(5):589–612, 2012.

[19] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 321–330, New York, NY, USA, 2011. ACM.

[20] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 335–344, New York, NY, USA, 2010. ACM.

[21] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Addison-Wesley, 2001.

[22] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 331–346, 2003.

[23] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 472–481, 2013.

[24] Wei Dong, Ji Wang, Xuan Qi, and Zhi-Chang Qi. Model checking uml statecharts. In *APSEC'01*, pages 363 – 370, 2001.

[25] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, 1999.

[26] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 7–16, 2010.

[27] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, October 2001.

[28] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 111–121, Washington, DC, USA, 2009. IEEE Computer Society.

[29] Leire Etxeberria and Goiuria Sagardui. Evaluation of quality attribute variability in software product families. In *ECBS*, pages 255–264, 2008.

[30] Leire Etxeberria and Goiuria Sagardui. Quality assessment in software product lines. In *High Confidence Software Reuse in Large Systems*, volume 5030, pages 178–181. 2008.

[31] M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE'12*, pages 573 –583, 2012.

[32] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *ICSE*, pages 341–350, 2011.

[33] Dario Fischbein and Sebastian Uchitel. On correct and complete strong merging of partial behaviour models. In *FSE'16*, pages 297–307, 2008.

[34] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, 2002.

[35] S. Gnesi, D. Latella, and M. Massink. Model checking uml statechart diagrams using jack. In *4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 46 –55, 1999.

[36] Steffen Göbel, Christoph Pohl, Simone Röttger, and Steffen Zschaler. The comquad component model: enabling dynamic selection of implementations by weaving non-functional aspects. AOSD '04, pages 74–82, 2004.

[37] N. Gold, A. Mohan, C. Knight, and M. Munro. Understanding service-oriented software. *Software, IEEE*, 21(2):71 – 77, march-april 2004.

[38] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley, 2004.

[39] Lars Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 31–40, New York, NY, USA, 2008. ACM.

[40] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. Param: A model checker for parametric markov models. In *CAV*, pages 660–664, 2010.

[41] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41:93–95, 2008.

[42] Svein Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using product line techniques to build adaptive systems. In *SPLC*, pages 141–150, 2006.

[43] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[44] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[45] George T. Heineman and William T. Councill. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[46] Thomas Henzinger, Shaz Qadeer, and Sriram Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV*, volume 1427, pages 440–451, 1998.

[47] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV*, pages 440–451, 1998.

[48] Anne Immonen and Eila Niemel. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7:49–65, 2008.

[49] S. Jarzabek, B. Yang, and S. Yoeun. Addressing quality attributes in domain analysis for product lines. *Software, IEE Proceedings -*, 153(2):61 – 73, 2006.

[50] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. In *Computer*, volume 36, pages 41–50, 2003.

[51] Dongsun Kim and Sooyong Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 0:76–85, 2009.

[52] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic model checking for performance and reliability analysis. *ACM Performance Evaluation Review*, 36(4):40–45, 2009.

[53] BeatrizPérez Lamancha, Macario Polo, and Mario Piattini. Systematic review on software product line testing. In *Software and Data Technologies*, volume 170 of *Communications in Computer and Information Science*, pages 58–71. Springer Berlin Heidelberg, 2013.

[54] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of uml statechart diagrams. In *FMOODS*, 1999.

[55] Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *ASE*, pages 269–280, 2009.

[56] Jaejoon Lee and Kyo C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. *International Software Product Line Conference*, 0:131–140, 2006.

[57] Lorea Belategi Leire Etxeberria, Goiuria Sagardui. Quality aware software product line engineering. In *Journal of the Brazilian Computer Society*, volume 14, pages 57 – 69, 2008.

[58] Rogerio Lemos and et al. Software engineering for self-adaptive systems: A second research roadmap. In Rogerio Lemos, Holger Giese, Hausi. Muller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. 2013.

[59] R. Lincke, T. Gutzmann, and W. Loewe. Software quality prediction models compared. In *10th International Conference on Quality Software (QSIC)*, pages 82 –91, 2010.

[60] Sten A. Lundesgaard, Arnor Solberg, Jon Oldevik, Robert France, JanOyvind Aagedal, and Frank Eliassen. Construction and execution of adaptable applications using an aspect-oriented and model driven approach. In *Proceedings of the 7th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS'07, pages 76–89, 2007.

[61] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*, pages 550–564. Springer-Verlag, 1992.

[62] MARTE. http://www.omgmarte.org/.

[63] Thomas Maßen and Horst Lichter. Requiline: A requirements engineering tool for software product lines. In *Software Product-Family Engineering*, volume 3014, pages 168–180. Springer Berlin Heidelberg, 2004.

[64] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37:56–64, July 2004.

[65] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[66] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ runtime to support dynamic adaptation. *Computer*, 42:44–51, October 2009.

[67] Eila Niemel and Anne Immonen. Capturing quality requirements of product family architecture. *Information and Software Technology*, 49:1107 – 1120, 2007.

[68] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54–62, 1999.

[69] Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. Unifying design and runtime software adaptation using aspect models. *Sci. Comput. Program.*, 76:1247–1260, 2011.

[70] Corina S. Pasareanu, Matthew B. Dwyer, and Michael Huth. Assume-guarantee model checking of software: A comparative case study. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 168–183, 1999.

[71] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

[72] Klaus Pohl, Gnter Bckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Heidelberg, Germany, 2005.

[73] C.M. Prashanth and K. Chandrashekhar Shet. Efficient algorithms for verification of uml statechart models. *Journal of Software*, 4(3), 2009.

[74] Genaina Rodrigues, David Rosenblum, and Sebastian Uchitel. Using scenarios to predict the reliability of concurrent component-based software systems. In *Fundamental Approaches to Software Engineering*, pages 111–126, 2005.

[75] Rick Salay, Marsha Chechik, and Jennifer Horkoff. Managing requirements uncertainty with partial models. In *20th IEEE International Requirements Engineering Conference (RE)*, pages 1 –10, sept. 2012.

[76] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In *FASE*, pages 224–239, 2012.

[77] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[78] P. Sampath, S. Arora, and S. Ramesh. Evolving specifications formally. In *RE'12*, pages 5–14, 2012.

[79] Ina Schaefer and Arnd Poetzsch-Heffter. Model-based verification of adaptive embedded systems under environment constraints. *SIGBED*, 6(3):9:1–9:4, 2009.

[80] P. Shaker, J.M. Atlee, and Shige Wang. A feature-oriented requirements modelling language. In *RE'12*, pages 151–160, 2012.

[81] N. Siegmund, M. Rosenmuller, C. Kastner, P.G. Giarrusso, S. Apel, and S.S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *15th International Software Product Line Conference (SPLC)*, pages 160 –169, 2011.

[82] N. Siegmund, M. Rosenmuller, M. Kuhlemann, C. Kastner, and G. Saake. Measuring non-functional properties in software product line for product derivation. In *15th Asia-Pacific Software Engineering Conference, APSEC '08*, pages 187 –194, 2008.

[83] J. Sincero, W. Schroder-Preikschat, and O. Spinczyk. Towards tool support for the configuration of non-functional properties in spls. In *42nd Hawaii International Conference on System Sciences, HICSS '09*, pages 1–7, 2009.

Bibliography

[84] J. Sincero, W. Schroder-Preikschat, and O. Spinczyk. Approaching non-functional properties of software product lines: Learning from products. In *17th Asia Pacific Software Engineering Conference (APSEC)*, pages 147 –155, 2010.

[85] Rasha Tawhid and Dorina C. Petriu. Integrating performance analysis in the model driven development of software product lines. In *MoDELS*, pages 490–504, 2008.

[86] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. Addison-Wesley, 2009.

[87] Pablo Trinidad, Antonio Ruiz Cortes, Joaquin Pena, and David Benavides. Mapping feature models onto component models to build dynamic software product lines. In *Workshop on Dynamic Software Product Lines*, pages 51–56, 2007.

[88] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behavior models from properties and scenarios. *TSE*, 35(3):384 –406, 2009.

[89] Sebastian Uchitel and Marsha Chechik. Merging partial behavioural models. In *FSE'12*, pages 43–52, 2004.

[90] E.S.K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*, pages 226–235, 1997.

[91] Hongyu Zhang, Stan Jarzabek, and Bo Yang. Quality prediction and assessment for product lines. CAiSE'03, pages 681–695, 2003.

[92] Qianchuan Zhao and Bruce H. Krogh. Formal verification of statecharts using finite-state model checkers. *IEEE Transactions on Control Systems Technology*, 14(5):943–950, 2006.

[93] Tewfik Ziadi, Loc Hlout, and Jean-Marc Jezequel. Towards a uml profile for software product lines. In *Software Product-Family Engineering*, pages 129–139. Springer LNCS, 2003.