Université de Namur, Belgique
Faculté d'informatique
Année académique 2013-2014

Model Checking for the Masses

Maxime Cordy

Thèse présentée en vue de l'obtention du grade de
Docteur en Sciences

Graphisme de couverture : © Presses universitaires de Namur

Rempart de la Vierge, 13
B - 5000 Namur (Belgique)

Imprimé en Belgique
ISBN : 978-2-87037-869-4
Dépôt légal: D/2014/1881/60

Jury

Prof. Joanne M. Atlee, University of Waterloo, Canada
Prof. Joel Greenyer, University of Hannover, Germany
Prof. Patrick Heymans, University of Namur, Belgium (co-advisor)
Dr. Axel Legay, INRIA Rennes, France
Prof. Michaël Petit, University of Namur, Belgium (chair)
Prof. Pierre-Yves Schobbens, University of Namur, Belgium (advisor)

Abstract

The model-checking problem for software product lines is harder than for single systems. Indeed, one has to verify all the software variants of a product line, whose number grows exponentially in the number of their differences. Techniques intended for single systems are inefficient in this case, for these can only be applied to all products separately. Variability-aware modelling formalisms and algorithms were designed as a more appropriate answer to that problem. Their strength lies in their ability to check a behaviour common to several products only once, which leads to substantial performance gains. Although they constitute a major step toward the efficient verification of product lines, these techniques are not yet mature enough to truly achieve this objective. They still lack optimisations to verify large models, the expressiveness to represent complex forms of variable behaviour, as well as usable languages and tools required for industrial transfer.

This thesis aims at improving product-line model checking in order to provide formalisms, algorithms, and tools that together hold the potential to verify real-world software product lines. We first study the state-of-the-art model-checking approaches for software product lines. We compare them in terms of available formalisms and algorithms, and we determine that the approach based on Featured Transition Systems (FTS) is the most suitable to act as a basis for the design of novel techniques. We then extend its theory along two axes: efficiency and expressiveness. For the former, we propose abstraction methods that can reduce the size of the model to check while maintaining correctness and completeness; they consist of extensions of single-system abstraction techniques applied to FTS, and of new techniques that abstract away from their variability. As for expressiveness, on the one hand we propose featured timed automata as a combination of FTS and real-time behaviour. On the other hand, we analyse how to support in FTS complex forms of variability such as numeric features and multi-features.

We implemented our theoretical results into a new model-checking tool named ProVeLines. As all our extensions are based on FTS, they share many commonalities. ProVeLines was therefore designed as a product line whose each variant implements a distinct combination of verification formalisms and algorithms. Finally, we show that the principles we designed for product-line model checking can also be applied to other formal methods, namely the verification of adaptive systems and the synthesis of controllers for product lines.

Keywords: Formal methods, model checking, software product line

Résumé

Vérifier un modèle de ligne de produits logiciels est plus difficile que vérifier un modèle d'un système unique. En effet, toutes les variantes d'une même ligne, dont le nombre croît exponentiellement par rapport au nombre de différences entre elles, doivent être vérifiées. Les techniques conçues pour les systèmes uniques sont inefficaces dans ce contexte, car elles ne peuvent s'appliquer qu'à un seul produit à la fois. Des formalismes de modélisation et des algorithmes spécifiques aux lignes de produits ont été conçus afin de répondre à ce problème. Ceux-ci vérifient un comportement exécuté dans plusieurs produits une seule fois, ce qui permet des gains de performance substantiels. Malgré que ces techniques constituent une avancée majeure, elles ne sont pas encore assez matures pour atteindre réellement leur objectif. Elles manquent encore d'optimisation pour vérifier de larges modèles, d'expressivité pour représenter des formes complexes de comportement variable, ainsi que de langages et d'outils nécessaires à un transfert industriel.

Cette thèse vise à améliorer des approches existantes afin de créer des formalismes, des algorithmes et des outils qui détiennent ensemble le potentiel de vérifier le comportement de lignes de produits industrielles. Nous étudions tout d'abord les techniques existantes et les comparons en termes de formalismes et d'algorithmes. Nous déterminons que l'approche basée sur les Featured Transition Systems (FTS) est la plus appropriée pour servir de base à la conception de nouvelles techniques. Nous étendons ensuite cette approche selon deux axes: efficacité et expressivité. Pour le premier, nous proposons des méthodes d'abstraction qui réduisent le modèle à vérifier tout en maintenant la correction et la complétude de la vérification ; ces méthodes comprennent tant des extensions de techniques existantes pour les systèmes uniques appliquées aux FTS que de nouvelles qui font abstraction de la variabilité. Quant à l'expressivité, d'une part nous définissons le featured timed automaton, un nouveau formalisme combinant les FTS avec du comportement temps réel ; d'autre part nous étudions comment étendre les FTS afin qu'ils supportent des formes complexes de variabilité telles que les features numériques et les multi-features.

Nous avons implémenté nos résultats théoriques dans un nouvel outil de model checking nommé ProVeLines. Comme toutes nos extensions sont basées sur les FTS, elles partagent de nombreuses similarités. Nous avons ainsi conçu ProVeLines comme une ligne de produits dont chaque variante implémente une théorie de vérification particulière. Enfin, nous montrons que les principes que nous avons établis pour la vérification de modèles peuvent également s'appliquer à d'autres méthodes formelles, en l'occurrence la vérification de systèmes adaptatifs et la synthèse de contrôleurs pour ligne de produits.

Mots-clés: Méthodes formelles, vérification de modèle, ligne de produits

Contents

Preface

Variability is ubiquitous in today's systems, be it in the form of configuration options or extensible architectures. By mastering variability, developers can adapt their system to changing requirements without having to develop entirely new applications. Software Product Lines (SPLs) are a popular form of variability-intensive systems. They are families of similar software systems developed together to make economies of scale [60, 149] and to reduce time to market. SPL engineering is a recent software engineering discipline that aims at facilitating the development of the members of the family (called *products* or *variants*). To achieve this objective, it advocates systematic and planned reuse, and encourages identifying *upfront* the commonalities and differences between these products. Variability in SPLs is commonly represented in terms of *features*, i.e. units of difference between products that appear natural to stakeholders. Each product is then defined by its set of features.

Nowadays, SPL engineering is widespread in industry, including in critical areas like automotive and avionics. The emergence and the increasing popularity of SPLs have raised the need for specific quality assurance techniques. Indeed, engineers have to provide solid evidence that *all* the products they build satisfy their intended requirements. Moreover, in case of failure they should identify which features or combinations of features are responsible for the errors in order to facilitate repair. Single-system quality assurance techniques are not appropriate to address this problem, for these can only be applied product by product. This is clearly suboptimal: variants of an SPL likely share lots of commonalities, which should be verified only once. Moreover, when the set of products is large, it is difficult to identify which (combinations of) features are responsible for breaches of quality. In the particular case of software verification, this identification process is essential, as errors may originate from unexpected interactions between features on which humans alone cannot have a comprehensive view.

Model checking is an established technique to verify a behavioural model of a system – typically a transition system – against a property expressed in temporal logic [45, 16]. It relies on an exhaustive exploration of the model in search for counterexamples, i.e. executions that violate the property to verify. Due to its exhaustiveness, model checking is costly in time and

memory. When applied to real systems with a typically huge state space, it faces a combinatorial blow-up called *state explosion*. The model-checking problem is even harder for SPLs: in this case, the model checker must either prove the absence of errors or find a counterexample for *each* variant that can produce a violation. As the SPL and its number of products grow, variability dramatically exacerbates the complexity of model checking. It is thus not feasible to apply single-system model checking to the thousands of variants that can compose real-world SPLs.

In recent years, many *variability-aware* techniques have been designed to address the SPL model-checking problem [103, 59, 58, 15]. These techniques keep track of variability information contained in an SPL behavioural model to associate each execution path to the exact set of variants able to produce it. By doing so, they are able to identify the set of products that violate a given property, and to report a counterexample of violation for each of them. Moreover, being aware of variability allows them to check behaviour common to several products only once. This is a clear improvement over an enumerative application of single-system model checking, which verifies a behaviour as many times as there are products that can exhibit it.

Problem Statement

Although earlier experiments suggest that these dedicated techniques bring substantial performance gains [56, 13], further improvements are required to verify industrial SPLs. First, SPL model-checking methods suffer from the state-explosion problem inherent to model checking, and their practical time complexity still grows more than linearly with the number of features [53]. Because of that, SPL model checkers perform better than mature, highly-optimised single-system model checkers only in specific cases, even though they explore fewer behaviours [56].

Second, current techniques are only applicable to SPL models with discrete behaviour, whereas critical systems typically have to meet other types of requirements such as real-time. Another of their fundamental limitations is that they do not deal with neither numeric features nor features appearing several times in a product, *viz.* multi-features (sometimes misleadingly referred to as *clones* in the literature [76, 142]). An example of a numeric feature could be *"maximum number of users"*, which can take different numeric values across the range of products. The multi-feature construct could be used to represent, e.g., processing units whose number is variable from one product to another and which can be configured independently. Despite evidence of their usefulness in practice, no SPL modelling tool currently supports multi-features [52], and SPL model checkers support neither numeric features nor multi-features. All these *expressiveness* limitations impede an industrial application of SPL verification.

Objectives

The above observations lead us to the central objective of this thesis: overcoming the aforementioned limitations to allow industrial applications of SPL model checking in the long term. This manuscript thus aims at providing theories to address a *variety* of verification problems on families composed of *lots* of systems, as well as tools that bridge the gap between these theories and engineers. Our ultimate goal is thus to enable *model checking for the masses*. We decompose this objective into three challenges.

O1. Improve the efficiency of SPL model-checking algorithms
Variability exacerbates the state-explosion problem inherent to model checking by adding a second source of complexity, *viz.* the number of products. In single-system verification, one of the most effective answers to state explosion is *model abstraction*, which creates more concise – therefore easier to verify – models of the system, typically by merging similar states. A fundamental challenge to address is the design of abstraction techniques that can reduce the two-dimensional complexity of SPL behavioural models. This challenge is mainly addressed in Part II.

O2. Increase the expressiveness of SPL behavioural formalisms
Most of the research in model checking is concerned with the verification of discrete behaviour against temporal properties. However, the discrete behaviour is often only a partial view of the system. In order to express and verify other requirements like real-time properties, new logics to express behaviour and extensions of the discrete formalisms were proposed together with verification algorithms. A first part of this second objective is to study if and how existing SPL verification techniques can be extended in the same way. A second part aims at overcoming their limitations in the expression of variability. Indeed, SPL model-checking approaches were designed to support Boolean features only, whereas recent surveys emphasized the need for additional forms of variability [115]. It is thus necessary to analyse the feasibility of equipping current approaches with the support of non-Boolean features. This objective is the main focus of Part III.

O3. Design of product-line verification tools
Our third objective is to make our theoretical results available in software tools. Single-system model checkers are generally equipped with a specification language used to describe the behaviour of a system to check, which can be textual (e.g. Promela in Spin [114], SMV in NuSMV [44], MRMC [121]) or graphical (e.g. UPPAAL [22], EmbeddedValidator [1], and Reactis Tester [152]). Although these may require too much expertise to be used directly by engineers, they can still act as cornerstones on the basis of which more usable languages are designed. Given the range of formalisms

we aim to define, we have to develop a common language to express all of them. In the end, we thus aim at providing a usable tool that, given an instance of this language and a property to verify, displays the SPL products that violate the property and is able to explain the violation.

Application to Other Formal Methods

Model checking is but one of the formal methods that can be used for quality assurance. Other domains could benefit from heuristics similar to those designed for FTS. Concretely, we focus on adaptive systems and controller synthesis. The challenge is to study these problems from a variability point of view and to provide theories and algorithms to solve them. This is our fourth and last objective.

O4. Design SPL-oriented methods for the verification of adaptive systems and the synthesis of controllers
Adaptive systems are "systems that are able to adjust their behaviour in response to their perception of the environment and the system itself" [42]. It means that these systems can be viewed as a set of systems, only one being active at a time. They can thus be considered as a *dynamic* product line. Our objective is to define a formal framework to model such systems, as well as to specify and verify properties on them. On the other hand, the product-line controller synthesis problem has received little attention and remains open. Although a few existing methods already provide partial solutions to that problem [99, 100], they are still insufficient. Ideally, the theory of controller synthesis should be generalized to product lines as model checking has been in the past years. This first requires a formal definition of that problem; then efficient algorithms to synthesize a controller for each product while maximizing reuse.

Review of SPL model-checking methods

A prerequisite to meet our objectives is the study of the existing SPL model-checking approaches. After a survey of the scientific literature, we identify three formalisms to model SPL behaviour:

1. Featured Transition Systems (FTS) [59, 58] associate transitions with constraints over the features, expressing that the products able to execute a transition are those whose features satisfy the corresponding constraint.

2. Modal Transition Systems (MTS) [116] are transition systems where transitions are either mandatory or optional. Optionality represents

transitions that are part of only a strict subset of the SPL, whereas mandatory transitions are executable by all products [90]. When MTS are combined with an additional logic named MHML [15], it is possible to associate an optional transition with the products that can execute it.

3. Multi-valued models [31] are a generalization of transition systems that defines transitions and properties over a multi-valued (as opposed to Boolean) logic. Thanks to an ad-hoc association between elements of this logic and set of products, multi-valued models are able to express variable behaviour [103].

After recapitulating the basic definitions of the above three formalisms, we compare them in terms of expressiveness and associated algorithms. We finally select FTS as the basis for our work. FTS were specifically designed to model product lines, and can be combined with formalisms to model variability such as feature models [119]. Moreover, the associated verification algorithms [56] exploit the variability information contained in the FTS in order to avoid performing redundant verifications. There exist no equivalent algorithms for multi-valued models and MTS. Moreover, the latter alone do not have a sufficient expressiveness to solve the SPL model-checking problem: they require an additional logic, *viz.* MHML, to be as expressive as FTS.

Contributions

To achieve the four aforementioned objectives, we propose a set of solutions that together constitute the contributions of the thesis.

C1. Abstraction methods for FTS
To increase the efficiency of SPL model checking, we design abstractions for SPL behavioural models. Abstraction often creates inaccuracies in the models, and thereby affects the properties they satisfy. A reported counterexample can therefore be *spurious*, that is, it exists within the abstract model but not in the real, concrete model. In this case, the abstraction must be refined to eliminate this false positive. Common methods to achieve this refinement make use of the spurious counterexample itself. They give rise to Counterexample Guided Abstraction Refinement (CEGAR), i.e. abstraction techniques that iteratively refine an abstract model until either they find a real counterexample or they can prove the absence of violation [48]. In spite of their success in single-system model checking, abstraction techniques have not yet been studied in the specific context of SPLs. We fill this gap and propose SPL-specific abstraction procedures based on CEGAR. As an SPL model is composed of a state space and variability information, we distin-

guish between *state abstraction* that only merges states as in single-model abstraction, *feature abstraction* that abstracts variability, and *mixed abstraction* that combines the previous two types. This latter type is the most complicated to implement, as spuriousness can originate from the merging of states, the abstraction of features, or both. We evaluate the efficiency of the three types of abstraction with respect to verification time.

C2. Variability-aware simulation relation
The correctness of abstraction with respect to property preservation is commonly proven on the basis of behavioural conformance relations such as simulation relations [144]. These, however, are not aware of variability. An essential prerequisite for reliable abstraction-based SPL model checking is thus the definition of a new simulation relation that can reason over a set of products. We study this relation and propose efficient algorithms to compute it. As for property verification, we want to exploit the commonality between the products behaviour to minimize the computation effort.

C3. Modelling and verification of real-time product lines
Timed automata [82] are a popular formalism for reasoning over the behaviour of continuous-time systems. The main reasons are that timed automata model-checking remains decidable for branching-time logics, and there exist efficient solutions to encode and manipulate real time. Moreover, renowned implementations like UPPAAL [23] managed to catch the interest of both academics and practitioners. This increases our confidence that timed automata are suitable in the context of verifying properties of real-time SPLs as well. Accordingly, we define featured timed automata, a new formalism that extends timed automata with variability in the same way as FTS extends transition systems. We also combine the principles of timed automata and FTS model checking into new algorithms that can verify real-time requirements.

C4. Support for multi-features and numeric features
Introducing multi-features is not as simple as it seems. Many semantics are possible [142], and we must define one formally before extending FTS to represent this kind of feature. Previous approaches define the semantics of multi-features as multi-sets [142]. However, since we are interested in behaviour we need to associate a behaviour to the exact feature instance responsible for it; multi-sets cannot provide this capability. Accordingly, we propose an array-based semantics for multi-features. With the addition of numeric features, optional behaviour can be made dependent not only on the presence or absence of functionality, but also on the satisfaction of arithmetic constraints over numeric parameters. Augmenting SPL model checking with numeric features requires a generalization of the underlying formalism, *viz.* FTS, and its associated algorithms. The need to handle

arithmetic constraints naturally leads us to use "Satisfiability Modulo Theory" (SMT) solvers. Still, these are more computationally expensive than solutions for purely Boolean satisfiability. Experiments we carried out show that while including multi-feature does not constitute a threat to performance, the use of SMT solvers drastically increases verification time. We thus also examine an alternative solution where attributes are converted into sets of Boolean variables.

C5. ProVeLines: a product-line of verifiers for SPLs
Our next contribution is the implementation of our results into usable tools. All of our extensions being based on the original FTS formalism, it appears natural to develop a single software package offering the capability to model and perform verifications on SPLs. This package is called ProVeLines. It allows a user to model discrete and real-time SPL behaviour in fPromela, a variability-aware extension of Promela originally designed for the SNIP model checker [53]. Given an fPromela model and its characteristics, ProVeLines checks properties of interest on the model using an appropriate algorithm, and then reports the set of products that satisfy the properties.

C6. Model checking adaptive systems within an SPL framework
We propose to see adaptive systems as systems with highly dynamic features. We model as features not only the reconfigurations of the system, but also the changes of the environment, such as failure modes. The resilience of the system can then be defined as its capability to select an adequate reconfiguration for each possible change of the environment. We must take into account that reconfiguration is often a major undertaking for the system: it has a high cost and it might make functions of the system unavailable for some time. Since these constraints are domain-specific, we provide a modelling formalism to specify them, and a property language to describe the requirements on the adaptive system. We design algorithms to determine how the system must reconfigure itself to satisfy its intended requirements.

C7. Mass synthesis of product-line variants
Scenario-based specification languages are an adequate means of intuitively yet formally specifying the behaviour of the system and its environment. Desired interactions being scattered across different scenarios, it rapidly becomes difficult to grasp what the system should do to ensure the satisfaction of these scenarios. The purpose of synthesis is to produce, from a scenario-based specification, a controller that drives the behaviour of the system whatever happens in the environment. An established method to solve the single-system synthesis problem is to consider it as a reachability game [78]. To transpose the above methodology to SPL engineering, we propose a method to specify the effects of features as a set of scenarios. We then extend the synthesis process by first defining a new type of reach-

ability game, which includes constructs to represent variability. Then we design algorithms to solve these novel reachability games, using heuristics similar to those developed for FTS. Thereby, we allow the synthesis of many controllers from similar specifications.

Structure

This manuscript is divided into the following six parts.

Part I recapitulates essential background. Chapter 1 introduces SPLs and the formal definitions of feature modelling. Chapter 2 recalls all the concepts related to single-system model checking that are needed to present the contributions of this thesis. Chapter 3 presents FTS and its companion theory and algorithms. Finally, Chapter 4 surveys the other approaches to formal verification of variability-intensive systems. It focuses more particularly on modal transition systems and multi-valued model checking, which are compared to FTS.

Part II is dedicated to FTS abstraction. Chapter 5 first introduces our variability-aware simulation relation as a means to compare the behaviour of two FTS. The usefulness of this relation is then illustrated in Chapter 6, where we design new CEGAR procedures for SPLs and three families of abstraction functions. We evaluate the efficiency of these abstractions through several experiments.

Part III introduces all the extensions to the FTS formalism we designed. Chapter 7 presents featured timed automata and shows how these can be verified by a combination of FTS and timed automata algorithms. Chapter 8 defines a semantics for SPLs with multi-features and numeric features, and discusses how the FTS theory can be extended to support them.

Part IV focuses on tools. It consists of only Chapter 9, which presents ProVeLines, our new model checker that implements the approaches introduced in this manuscript. It notably explain its features, architecture, input languages, and user interface.

Part V studies variability-aware extensions of other formal methods. More precisely, Chapter 10 focuses on adaptive systems, their modelling and their verification as dynamic SPLs. Chapter 11 formally defines the synthesis problem and presents a method to synthesize controllers for an SPL's products based on feature and scenario modelling.

Part VI is composed of Chapter 12 only, which reviews the developments presented in this thesis and emphasizes the perspectives they offer.

Publications

Several chapters of this thesis are based on the following published articles, of which we are the first author :

[64] *Simulation-Based Abstractions for Software Product-Line Model Checking*, in Proceedings of the 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pages 672–682; with Andreas Classen, Gilles Perrouin, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. (Chapter 5)

[63] *Managing Evolution in Software Product Lines: A Model-Checking Perspective*, in Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS), ACM, 2012, pages 183–191; with Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. (Chapter 5)

[62] *Counterexample Guided Abstraction Refinement of Product-Line Behavioural Models*, in Proceedings of the 22nd International Symposium on Foundations of Software Engineering (to appear), ACM, 2014; with Maxime Cordy, Patrick Heymans, Axel Legay, Pierre-Yves Schobbens, Bruno Dawagne, and Martin Leucker. (Chapter 6)

[65] *Behavioural Modelling and Verification of Real-Time Software Product Lines*, in Proceedings of the 16th International Software Product Line Conference (SPLC), ACM, 2012, pages 183–191; with Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. (Chapter 7)

[68] *Beyond Boolean Product-Line Model Checking: Dealing with Feature Attributes and Multi-Features*, in Proceedings of the 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pages 472–481; with Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. (Chapter 8)

[69] *ProVeLines: A product-line of Verifiers for Software Product Lines*, in Proceedings of the 17th International Software Product Line Conference (SPLC), Volume 2, ACM, 2013, pages 141–146; with Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. (Chapter 9)

[62] *Model Checking Adaptive Software with Featured Transition Systems*, in Assurances for Self-Adaptive Systems, Springer, 2013, pages 1–29; with Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. (Chapter 10)

Acknowledgments

First, I would like to thank my advisors Pierre-Yves Schobbens and Patrick Heymans. They form an incredible duo, coming from different worlds with complementary abilities and mindsets. They largely contributed to shape the researcher that I am, having sharpened my scientific, analytic, and writing skills alike for three years. They are also two of the most interesting people I have met in my life. Working under their day-to-day supervision was truly an immense pleasure.

In addition to my advisors, I have to thank collaborators whose meeting had a significant impact on this thesis: Andreas Classen for trusting me and introducing me to this area of research when I was still a Master student; Axel Legay for his dedication to my work during these last three years and the numerous (non-)scientific discussions we had; Amir Molzam Sharifloo, the smallest and funniest PhD student I have met so far; Joel Greenyer for the nice time spent in Milan and the successful collaboration that followed; Gilles Perrouin for being such a gentle senior and allowing me to make fun of him from time to time. I am also grateful to my other colleagues from the University of Namur, which contributed to the nice atmosphere in which I performed my daily work.

This thesis was funded by the National Fund for Scientific Research (FNRS), through a research fellow grant, Project FC 91490. This grant allowed me to attend international conferences at different corners of the world. The travels, the venues, and the people I met altogether are an invaluable experience to me, which will leave unfading memories.

On the personal side, I want to thank my family and close friends, just for making life enjoyable. I thank Marine Leroi for loving the somewhat strange guy that I am and coping with all my particularities, for understanding how much this thesis meant to me, and for all the time spent together for the last eight years. Finally, I dedicate this manuscript to my mother, Solange Harzée, and my late father, Michel Cordy. Since my youngest age, they taught me how important it is to exploit my abilities to their fullest potential. I know that the success of my PhD studies was an important source of pride for them. I will never be able to pay them back for all they did for me during these 25 years. Still, I hope that they will accept this work as a humble expression of my utmost gratitude. Thank you.

Part I

Background

Chapter 1

Software Product Lines

The characteristics of today's market are the results of many mutations in both producers' and customers' habits. A first key economic factor is the increasing capability of customers to afford to buy various kinds of products. Producers have thus to deal with an increasing demand while avoiding delays in delivery. In order not to be overwhelmed, manufacturers abandoned the individual handcraft approach and adopted the production lines, which consist in executing sequential operations during which materials are systematically combined to form a final product. This new production paradigm successfully enables mass-production for reduced costs in many industries such as automotive, alimentary, and metallurgy.

Despite the economic benefits of production lines, manufacturers are now confronted to new market trends they cannot deal with. Indeed, customers desire to buy unique products tailored to their specific tastes and needs. The systematic nature of production lines impedes producers to enable diversification, and thus to meet customers' variable requirements. These observations led industries to revisit the notion of production lines into what is called *mass customisation*.

Definition 1.1 (Mass customisation)
[79] Mass customisation is the large-scale production of goods tailored to individual customers' needs.

The most prominent example of mass customisation is arguably the automotive industry. Any car constructor offers a broad range of different car models, each of which can be further customized through the selection of many options. This allows customers to configure and purchase the car that suits them the best. To cope with both the diversity and the huge demand, car manufacturers invested in common development platforms wherein they

build different types of cars. By doing so, they managed to make economies of scale while rapidly delivering customized products.

1.1 Software Mass Customisation

The software industry being way younger than automotive or metallurgy, it had not encountered the need for mass customization until the recent years. Even to this day, most of the developed software can be categorized into general-purpose applications software (e.g., word processing, database, and graphic applications) meant to target a broad market segment, and custom-made applications designed for a specific customer. The former kind offers a nice return on investment for software-developing companies, but it lacks a sufficient diversification. On the contrary, individual software are tailored to customers' specific requirements but are expensive to produce and to maintain. In order to increase the scope of the market segment they target, software engineers have sought to bring the principles of mass customisation into their discipline.

This is, however, more challenging that it looks. Unlike other industries, software can hardly be regarded as a simple combination of components. Indeed, a lesson learned from the software crisis in the 1960s is that software consist not only of compiled source code, but also of elicited requirements, an architecture, quality assurance procedures, documentation, etc. The slightest modification in customers' requirements can impact all these ingredients, and thus implies changes that cross-cut the whole development life cycle. Allowing mass customisation thus obliges software engineers to rethink their development processes. This resulted in a new software engineering paradigm called *software product-line engineering*.

Definition 1.2 (Software product line)
[60] A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

The above definition highlights three important characteristics of SPL development:

1. The *features* shared by the software products (also called *variants*) have to be identified and managed.

2. Final software products are built from *reusable* assets (as opposed to being developed from scratch)

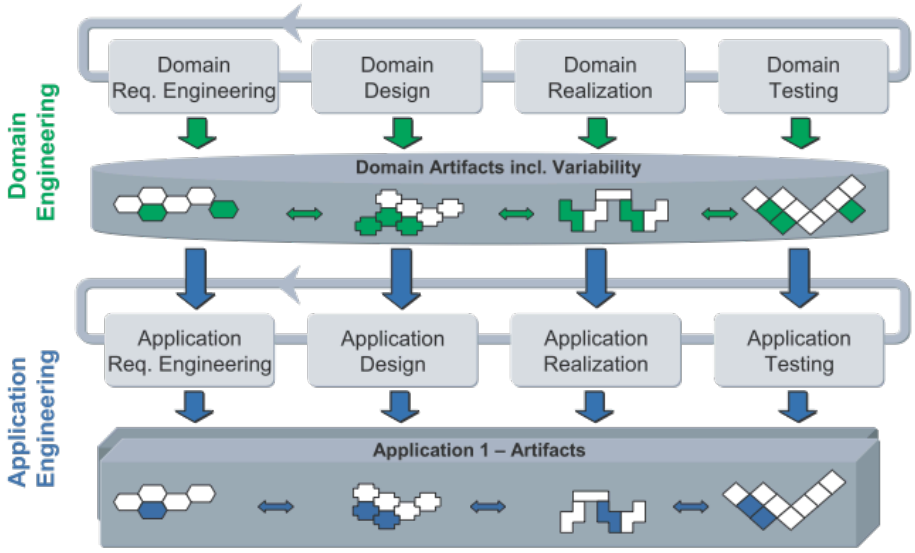


Figure 1.1: The SPL development framework [149]

3. If necessary, these assets are tailored through *preplanned* variation mechanisms.

Building a new product thus comes down to assembling the appropriate variations of the core assets. Accordingly, the development of SPLs (see Figure 1.1) involves two distinct processes: *domain* engineering and *application* engineering [149]. During domain engineering, the commonality and the variability between the products is identified, and different types of reusable artefacts (which can be requirements, design, code, tests, etc.) are built. During application engineering, engineers exploit the variability information to derive products from the appropriate (variations of) assets. Assuming a fixed set of features, domain artefacts and their variations are intended to be built once and for all. The complexity of developing a set of products is thus concentrated on a single development cycle, whereas single-system software engineering would require as many separate cycles as there are variants to build. This is the reason why adopting SPL engineering to develop a families of products is substantially more time- and cost-efficient [60].

However, these benefits come at the cost of a new source of complexity: variability has to be managed throughout the whole development life cycle. This requires an initial activity where the commonality and the variability are identified, as well as the creation and the maintenance of traceability links between variation points and domain artefacts. In SPL engineering, variability is typically expressed in terms of *features*. Features received many definitions in the past [57]. In this thesis, we consider features as

units of difference between products that appear natural to stakeholders and technicians alike [57]. An SPL variant is thus defined as a subset of the SPL features. Classen *et al.* [57] gives more formal, requirements-engineering oriented definition of feature based on Zave and Jackson's framework [165].

Definition 1.3 (Feature)
[57] A feature is a triplet $f = (S,W,R)$ where S is the specification of the feature, W is the factual assumptions the feature makes about its environment, and R is the requirements the feature should satisfy.

We assume that features are consistent: provided that the assumptions W hold the specification S of the feature ensures that the requirements R are satisfied, noted $S, W \vdash R$ in Zave and Jackson's framework. The consistency of the individual features does not imply that any SPL product is consistent, though. Indeed, undesired *feature interactions* may occur and modify the expected effects of features, thereby preventing the fulfilment of requirements yet satisfied by the individual features. *Invalid* products resulting from interacting features thus do not behave properly, as one would expect them to meet all the requirements of their constituent features as well as some new requirements defining a proper interaction. They should therefore not be build in order to prevent unpredicted inconsistencies. Feature interactions are certainly not the only source of product invalidity. Two features may be dependent on each other for various reasons, which can result in one requiring or excluding the other. Products may also be excluded from the product line for other purposes, e.g. economical reasons. More generally, there is a need to define dependencies between features to prevent disallowed combinations. *Variability modelling* is the activity of eliciting the features of an SPL and their inter-dependencies. These are commonly represented in a modelling language, and then named feature models [119].

1.2 Feature Models: Syntax and Semantics

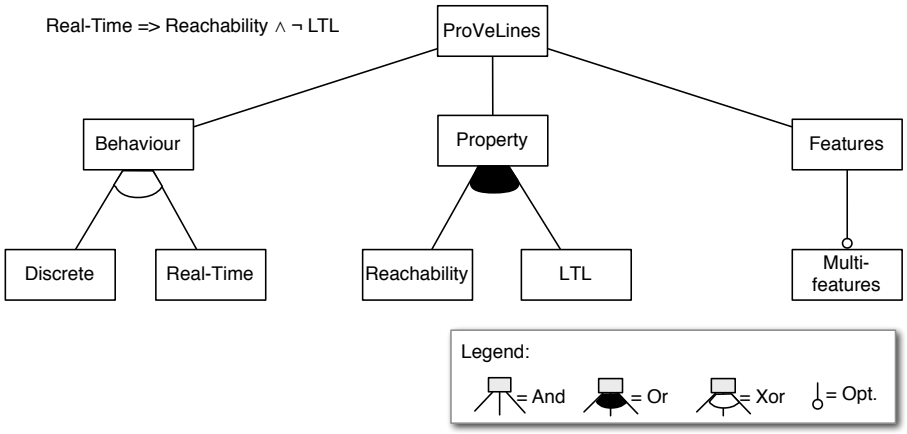
A Feature Model (FM) is a hierarchical structure that defines relations between features. Originally, they were given a graphical form [119] but many textual languages [157, 11, 17, 21, 25, 52] have been proposed as alternatives. We focus on graphical FMs and TVL, one of the latest incarnations of textual FM, due to some of its advantages: high expressiveness, formal semantics and tool support [52].

Figure 1.2a shows a TVL model.[1] The equivalent graphical representa-

[1] This FM actually models a subset of the variability of ProVeLines, our product line

```
root ProVeLines group allOf {
    Behaviour group oneOf {
      Discrete,
      Real-Time
    },
    Property group someOf {
      Reachability,
      LTL
    },
    Features group allOf {
      opt Multi-features
    }
    Real-Time => Reachability ∧ ¬ LTL
}
```

(a) Written in TVL



(b) Shown in graphical form

Figure 1.2: An example of feature model

tion appears on Figure 1.2b. In both representations, the FM's fundamental structure remains a tree that reflects the parent-child hierarchy between the features. At the top of the tree lies the *root* feature (`ProVeLines`). The root is always part of a product, regardless of the other features. A feature may have *child features*; for instance, `ProVeLines` has three: `Behaviour`, `Property`, and `Features`. Parent-child relationship have a decomposition type among *and*, *or*, and *xor* (named *allOf*, *someOf*, and *oneOf* in TVL). These types respectively specify that a parent feature must have all, at least one, or only one child feature in a given product. When the decomposition type is *and* a child feature can be labelled as optional, meaning that this feature is not mandatory.

Since they were first introduced by Kang *et al.* in 1990 [119], FMs became more sophisticated (see [154, 52] for detailed surveys of FM languages). Among the most intensively used extensions, one finds *group cardinalities*. These are a generalization of the decomposition types, which defines the minimum and maximum number of children a feature may have in any product. In our example, the group cardinality of feature `Property` is $[1, 3]$. Feature attributes and feature cardinalities are two other extensions that have received a lot of attention in the recent years [74, 142]. Since their presence implies drastic changes in FM semantics, we do not consider them at this point. We introduce them properly in Chapter 8.

The tree structure alone may not always be sufficient to model all the dependencies between features. Therefore, recent FM languages like TVL permit the definition of additional constraints (i.e. propositional formulae) over the features. In Figure 1.2, one such constraint specifies that feature `Real-time` requires `Reachability` and excludes `LTL`. The addition of a constraint language makes FMs as expressive as Boolean logic when it comes to expressing constraints over the features.

Now that the FM constructs have been informally introduced, we give them an abstract syntax which further serves as a basis for the definition of a formal semantics. This abstract syntax remains valid for most feature modelling languages without feature cardinalities and attributes.

Definition 1.4 (Feature model)
Let F be a set of features. An FM over F is a tuple $(F, r, DE, \lambda, \Phi)$ where:

- *F is a non-empty, finite set of features.*
- *$r \in F$ is the root.*

of SPL model checkers (see more in Chapter 9).

- $DE \subseteq F \times F$ *is the decomposition (hierarchy) relation between features. For any* $(f, f') \in DE$, f *is the parent and* f' *is the child feature. By* $children(f)$ *we denote the set of child features of* f.
- $\lambda : F \to \mathbb{N} \times (\mathbb{N} \cup \{*\})$ *indicates the decomposition operator of a feature, i.e. its group cardinality. If* $\lambda(f) = (n, m)$ *then* n *is the minimum group cardinality of* f *and* m *its maximum group cardinality. If* $m = *$ *then the number of selected children can be any finite value not less than* n.
- Φ *is a Boolean formula over* F, *expressing additional constraints.*

Furthermore, any FM must satisfy the following well-formedness rules:

- r *has no parent:* $\nexists f \in F \bullet (f, r) \in DE$;
- *DE is acyclic:* $\nexists f_1, \ldots, f_k \in F^k \bullet (f_k, f_1) \in DE \wedge \forall i \in \{1, \ldots, k-1\} \bullet (f_i, f_{i+1}) \in DE$.

In the original definition of FM [119], a feature can be labelled as optional. Our abstract syntax intentionally gives up this capability. The reason is that this construct may cause confusion, especially but not only when combined with group cardinalities. Many existing tools therefore support optional features only in *and* decompositions [52], the only case where optionality has a useful meaning. In FMs without group cardinalities, Czarnecki and Eisenecker [75] define that optionality has priority over the decomposition type. This implies that a parent feature with a *xor* or an *or* decomposition type may have no child feature. Intuitively, this happens when the sole selected child feature is optional, and this feature is not present in the product. The decomposition type is satisfied in the sense that one feature has indeed been selected, even though it is not included in the product in the end due to its optional nature. Optionality takes an even more subtle form in case of group cardinalities. Leaning on Czarnecki and Eisenecker's work, Classen *et al.* [52] generalize the impact of the "optional" label as follows: optionality causes the lower bound of a group cardinality to decrease by the number of optional child features in the decomposition, and these are not counted to determine the satisfaction of the lower bound. Although this convention gets rid of all the ambiguities related to optionality, we believe that they remain counter-intuitive in the eyes of FM users. This is the reason why our abstract syntax does not support the "optional" label; all features are optional by default. This restriction, however, does not reduce the expressiveness of FM since group cardinalities and the additional constraints Φ can be used to simulate (non-)optionality. If we except feature attributes and

feature cardinalities, our abstract syntax is thus valid for the most common FM modelling languages surveyed in [52].

An FM represents a set of constraints over features. Thereby, it specifies which combinations of features can be included in products. The semantics of an FM is accordingly defined as the set of valid products, i.e. the sets of features satisfying the constraints expressed by the FM [154].

Definition 1.5 (Feature model semantics)
Let $fm = (F, r, DE, \lambda, \Phi)$ *be an FM. The semantics of fm, noted* $[\![fm]\!]$*, is the largest set of products such that* $\forall p \in [\![fm]\!] \subseteq 2^F$*, we have:*

- *p contains the root:* $r \in p$*;*
- *p respects the parent-child relationship:*

$$g \in p \wedge (f, g) \in DE \Rightarrow f \in p;$$

- *p satisfies the group cardinalities:*

$$f \in p \wedge \lambda(f) = (n, m) \Rightarrow n \leq |\{g \in p \mid (f, g) \in DE\}| \leq m;$$

- *p satisfies the additional constraints:* $\bigwedge_{f \in p} f \bigwedge_{g \in F \setminus p} \neg g \models \Phi$.

Then fm is consistent if and only if $|[\![fm]\!]| > 0$.

Let us illustrate the above abstract syntax and semantics through the FM of Figure 1.2.

Example 1.2.1 *The FM of Figure 1.2 consists of nine features, among which* $ProVeLines$ *is the root. The parent child relationship* DE *is given as*

$$\{(ProVeLines, Behaviour), (ProVeLines, Property), (ProVeLines, Features), (Behaviour, Discrete), (Behaviour, Real\text{-}time), (Property, Reachability), (Property, LTL), (Features, Multi-features)\}.$$

The group cardinality of the leaves $Discrete$*,* $Real\text{-}time$*,* $Reachability$*,* LTL*, and* $Multi\text{-}features$ *is* $(0, 0)$*. As for the others, we have*

$$\begin{aligned} \lambda(ProVeLines) &= (3, 3) \\ \lambda(Behaviour) &= (1, 1) \\ \lambda(Property) &= (1, 2) \\ \lambda(Features) &= (0, 1). \end{aligned}$$

Table 1.1: Semantics of the FM shown in Figure 1.2 (✓ is "yes", ✗ is "no', – is "any")

PVL	Beh.	Pro.	Fea.	Dis.	R.-t.	Rea.	LTL	M.-f.	**Valid**	/512
✗	–	–	–	–	–	–	–	–	✗	256
✓	✗	–	–	–	–	–	–	–	✗	128
✓	✓	✗	–	–	–	–	–	–	✗	64
✓	✓	✓	✗	–	–	–	–	–	✗	32
✓	✓	✓	✓	✓	✓	–	–	–	✗	8
✓	✓	✓	✓	✗	✗	–	–	–	✗	8
✓	✓	✓	✓	✓	✗	✗	✗	–	✗	2
✓	✓	✓	✓	✗	✓	✗	–	–	✗	4
✓	✓	✓	✓	✗	✓	✓	✓	–	✗	2
✓	✓	✓	✓	✓	✗	✓	–	–	✓	4
✓	✓	✓	✓	✓	✗	✗	✓	–	✓	2
✓	✓	✓	✓	✗	✓	✓	✗	–	✓	2

Finally, the additional constraints Φ *consist of only one implication:*

$$\Phi = (\mathit{Real\text{-}time} \Rightarrow \mathit{Reachability} \wedge \neg \mathit{LTL}).$$

According to Definition 1.5, 8 valid products are derivable from this FM; see Table 1.1.

One can observe that Table 1.1 is a (compressed) truth table. As mentioned before, FMs can indeed be regarded as Boolean formulae. This observation illustrates the basis for modelling traceability links between features and core asset variations, as detailed in the following section.

1.3 Variability Encoding

The development of core assets follows the identification of SPL features and their inter-relations. Each asset of an SPL is actually more than its single-system counterpart, for it can be derived into several variants. Regardless of the asset type, it is essential to create and maintain a link between a given variation and the features it corresponds to. Instead of developing every variation separately, a core asset is typically accompanied with information describing the effect that a given (combination of) feature(s) has on the asset. One commonly distinguishes two variability-modelling paradigms: *annotative* and *compositional* [120].

The annotative paradigm consists in building a core asset that contains information about all products, and in associating variable parts of this

assets with the set of products that include this specific part. The asset variant specific to a given product p is then obtained by pruning parts of the core asset whose associated set does not include p. The advantage of this paradigm is that the effects of features appear on the asset directly. However, it does not allow a modular definition of features and therefore may loose engineers in an exponentially increasing complexity. This drawback can be alleviated by tools that can focus on a subset of products, or highlight the parts that are specific to this subset.

Labelling variable parts with an explicit set of product would create large annotations. In order to keep them concise, this set of products is commonly represented by a Boolean formula over the features, which we name *feature expression*.

Definition 1.6 (Feature expression)
Let $F = \{f_1, \dots, f_{|F|}\}$ be a set of features. Then a feature expression over F is a Boolean formula $b \in 2^{2^F}$ whose each variable corresponds to a unique element of F, and whose semantics is a function $2^F \to \{\bot, \top\}$ encoding a set of products. A product $p \in 2^F$ is included in the set represented by b, noted $[\![b]\!]$, if and only if $b(p) = \top$, or equivalently:

$$\bigwedge_{f \in p} f \bigwedge_{g \in F \setminus p} \neg g \models b.$$

In this case, p is said to satisfy *b, noted $p \models b$. We also denote by $\mathbb{B}(px)$ the feature expression encoding the set of products px, that is,*

$$\mathbb{B}(px) = \bigvee_{p \in px} \Big(\bigwedge_{f \in p} f \bigwedge_{g \in F \setminus p} \neg g \Big).$$

Given that their semantics is a set of products, FMs can also be encoded into a feature expression. FM analysis tools exploit that property to encode all the constraints expressed by an FM into a Boolean formula that a satisfiability solver can check, thereby allowing one to check desired properties like no-emptiness, and supporting configuration operations [20].

As opposed to annotative methods, the compositional paradigm consists in defining separately how each feature alters the core asset, which itself contains no form of variability. The asset variant related to a given product is then obtained by successively applying the effects of this product's constituent features. The possibility to specify features modularly is a clear improvement over the annotative paradigm. However, engineers may lose sight of subtle feature interactions that would be grasped thanks to the

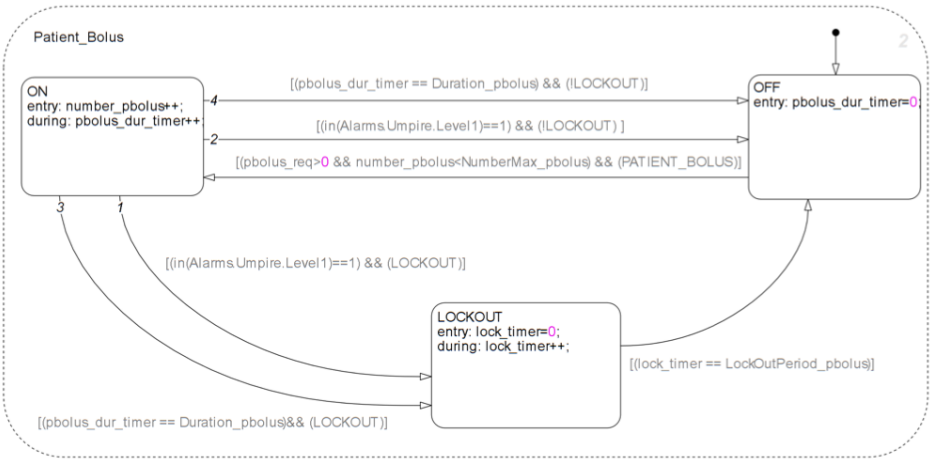


Figure 1.3: A Stateflow model annotated with features

complete view offered by an annotative method, although this can be alleviated by tools that compute the annotative form from the compositional one. Moreover, the effects of feature interactions may depend on the composition order. For example, assume two features triggered by the same event and with contradictory behaviour. In this case, the feature applied last may erase the effect of the other. An unavoidable consequence is that one cannot define a product as a set of features; instead it should be a *list* of features. Not only this increases the number of derivable variants, but it also introduces a new source of complexity. A solution to overcome that problem is to specify conflict resolution rules as part of the composition operator [155]. Still, this requires engineers to fully comprehend the rules imposed by this composition operator.

The two modelling paradigms hold their own strengths and weaknesses, which are often opposite. Still it is generally admitted that both have an interest in specific situations [120]. We therefore do not argue in favour of one or the other. The models we present in this thesis are for the most part based on the annotative paradigm. Moreover, compositional description can always be translated into an annotative equivalent; the converse is not true. For these reasons, we focus mainly on the annotative kind. We end this discussion with an instantiation of the paradigms in the context of behaviour modelling.

Example 1.3.1 *State machines are a common modelling language to represent software behaviour. In the case of SPLs, designers have to model the behaviour of several products at once. To that aim, they can build either a state machine containing all the information about the products behaviour, or an initial state machine combined with individual description of how features modify the behaviour of the initial state machine.*

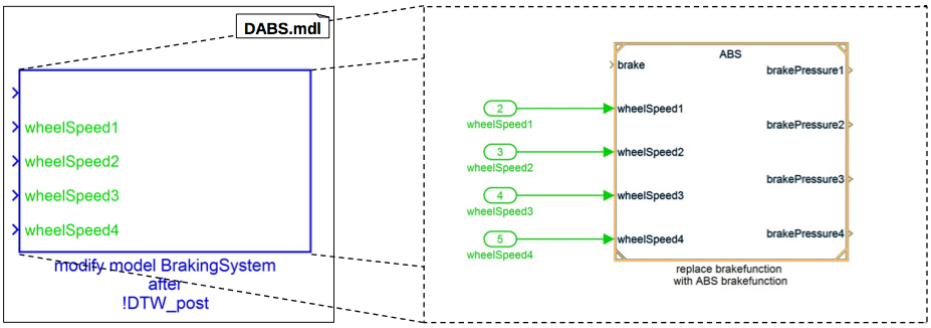


Figure 1.4: A compositional extension of Simulink based on delta modelling

A first approach to build variability-aware state machines consists in associating their transitions with a feature expression that defines which products can execute this transition. This approach has notably been applied to represent variability within stateflow [117] and transition systems [59, 58] (see more in Chapter 3). For example, Figure 1.3 shows a variability-aware stateflow model, where features are written in capital letters in order to distinguish them from other variables. Modelling variable behaviour in such languages is relatively straightforward as it requires neither ad-hoc constructs nor additional languages, provided that they already include Boolean expressions and conditionals.

On the contrary, Figure 1.4 shows a compositional extension of Simulink based on delta modelling [107]. The basic Simulink component `ABS` *lies on the right side. To the left, there is a delta module* `DABS` *specifying the modifications imposed on* `ABS` *when a certain feature expression ($\neg DTW\_post$) is satisfied. In this modelling fashion, the basic model completely ignores the existence of the delta modules, and these need information about only the parts of the model they actually modify.*

Chapter 2

Model Checking

SPL engineering is nowadays well-anchored in software industry, including in embedded critical systems like automotive, avionics, and medical systems. It is therefore essential to provide engineers with quality assurance techniques allowing them to prove that their software satisfy their intended requirements and contain no error.

Most of the quality assurance activities held in industrial development are based on testing. It consists in writing pieces of code that execute the existing code base to discover errors. The popularity of this kind of techniques originates mainly from the fact that it can be performed by any software developer. Moreover, it can be applied directly on the application (see the *Application Testing* phase in Figure 1.1), which makes easier to link a discovered error to the concrete part of the application responsible for it. However, testing suffers from two major disadvantages. First, it is only a *partial* verification methods whose scope is limited to the sole execution of the system triggered by the test itself. Second, design errors discovered at such a latter stage of development can require costly iterations to be fixed.

As opposed to testing, model checking [45, 16] is a *complete* verification technique for hardware and software. Given a model of the system and a property to check, a model checker exhaustively explores the possible executions of the model in search for violations of the property. If it finds one, it returns an example of execution that triggers the violation. Due to its exhaustive nature, model checking suffers from the state-explosion problem, i.e. verification becomes exponentially more costly as the size of the verified model increases. For that reason, it is in general infeasible to apply this technique directly on the application, as it can produce a huge number of executions. Instead, model checking is typically performed on models of the system (see *Application Design* phase in Figure 1.1). Because of that, it cannot guarantee that the yet-to-be-built application will be error-free. One can thus regard testing and model checking as complementary techniques rather than opposing them.

Since its introduction in the 1980s, model checking has evolved into a broad range of formalisms, algorithms, and tools. Whatever the specific context in which it occurs, the model-checking problem can be given the following general definition.

Definition 2.1 (Model checking)
Let M be a model of a system and Φ be a property. Model checking M against Φ is the problem of determining whether all the behaviours of M satisfy Φ, noted $M \models \Phi$, and if that is not the case, providing a counterexample of behaviour that violates Φ.

2.1 Modelling Behaviour as Transition Systems

There exist many modelling languages to express behaviour. The most popular find their origin in Harel's statecharts [109], which have been derived into many variants including UML state machines and Stateflow. Other kinds of language exist: process algebrae, sequence diagrams, Petri nets, and many others. Given the plethora of language variants, the model-checking theory relies instead on a fundamental formalism, typically a *Transition System* (TS). This is basically a directed graph whose nodes are states of the system and edges are transitions between states which model how the system can evolve from state to state. TS are given many definitions in the literature. In this thesis, we consider that *atomic propositions* define essential characteristics that may hold in a state of the system. These propositions are defined specifically according to the application domain. Also, transitions are labelled with *actions* whose main purpose is to give a meaningful name to transition. They are also a means of synchronisation between systems running in parallel [16], although we do not focus on that mechanism. Formally, TS are defined as follows.

Definition 2.2 (Transition system)
[16] A TS is a tuple $(S, Act, Trans, I, AP, L)$ where

- *S is a set of states named the state space;*
- *Act is a set of actions;*
- *$Trans \subseteq S \times Act \times S$ is the transition relation, where $(s, \alpha, s') \in Trans$ (also noted $s \xrightarrow{\alpha} s'$) means that there is a transition from state s to state s' labelled with action α;*

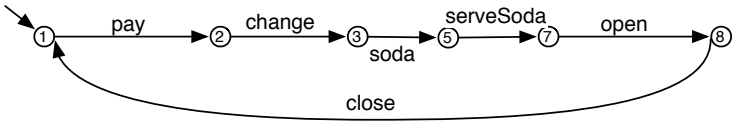


Figure 2.1: A TS modelling a soda vending machine

- $I \subseteq S$ *is a set of initial states;*
- AP *is a set of atomic propositions;*
- $L : S \to 2^{AP}$ *is a function that associates every state with the set of atomic propositions satisfied by this state.*

We assume that our TS have no terminal state, i.e. a state without outgoing transition: $\forall s \in S \bullet \exists \alpha \in Act, s' \in S \bullet (s, \alpha, s') \in Trans$.

From a model-checking perspective, TS act as a unified semantics for any behaviour modelling language. Every behavioural model is thus seen as a set of states and transitions. If we consider statecharts, the relation between such a model and the underlying TS is rather immediate: a TS state represents the set of active states in the statechart, and transitions are enabled according to the evaluation of their guard and state activation conditions. For Petri nets, it is also required to consider the number of tokens in every state. Defining the TS semantics is more difficult for languages that are not based on the concept of state. In the case of sequence diagrams, the cartesian product of the current positions in all the diagrams forms the state space of the underlying TS, whereas the transition relation is defined by the synchronisation of events labelling the next message of the diagrams. Still, TS have been recognized as the standard formalism to represent a system behaviour at a fundamental level.

Example 2.1.1 *Figure 2.1 shows a TS representing a soda vending machine [50]. This TS is composed of six states, each of which has a single outgoing transition leading to a subsequent state. The initial state, marked by an edge without origin, is state 1. The action labelling a given transition is shown on top of it. For instance, the action labelling the transition from state 1 to state 2 is named* `pay` *and models that a customer inserts a coin in the vending machine. The machine then gives the change (see transition from state 2 to state 3). The customer selects a soda beverage (transition from state 3 to state 5) before the machine serves it (transition from state 5 to state 7). Finally, the customer opens the compartment to get the drink*

(transition from state 7 to state 8) and the machine returns to its initial state (transition from state 8 to state 1). For readability, we did not display the atomic propositions satisfied by each state.

An *execution* of a TS is an alternating sequence of states and actions that starts from an initial state and satisfies the transition relation, *i.e.* an infinite sequence $s_0\alpha_0 s_1\alpha_1 \ldots$ with $s_0 \in I$ and $s_i \xrightarrow{\alpha_i} s_{i+1}$ for any $i \geq 0$. Intuitively, the set of executions represents the transitions that may occur in the TS. It is thus a discrete representation of the behaviour of the modelled system. Since we aim at verifying properties, we are interested only in what can affect the satisfaction of these properties. According to Definition 2.2, the atomic properties – the basic ingredient to specify arbitrarily complex behavioural properties – are defined over states. Therefore, the actions triggered during the executions have no importance in the verification process. Instead of the executions, we can thus consider the *paths* of the TS, that is, the infinite sequences of the form $s_0, s_1, \ldots$ where $s_0 \in I$ and $\forall i \geq 0 \bullet \exists \alpha_i \bullet s_i \xrightarrow{\alpha_i} s_{i+1}$. Moreover, it is not so much the visited states as the atomic propositions they satisfy that determine the satisfaction of properties. In other words, two distinct states satisfying the same set of atomic propositions are considered as equivalent. We define accordingly the semantics of a TS as a set of traces, i.e. sequences of atomic propositions satisfied during one of its executions.[1]

Definition 2.3 (Transition system semantics)
[16] Let $ts = (S, Act, Trans, I, AP, L)$ be a TS. The semantics of ts, noted $[\![ts]\!]$, is its set of traces:

$$[\![ts]\!] = \{L(s_0), L(s_1), \cdots \in (2^{AP})^\omega \mid s_0 \in I \wedge \forall i \in \mathbb{N} \bullet \exists \alpha_i \in Act \bullet (s_i \xrightarrow{\alpha_i} s_{i+1})\}.$$

For example, the semantics of the TS shown in Figure 2.1 is a singleton whose element is the sequence of atomic propositions satisfied when running infinitely often in the unique loop of the TS, that is,

$$\big(L(1)L(2)L(3)L(5)L(7)L(9)\big)^\omega.$$

[1] This definition is appropriate for linear logics such as the one we present in the next section. Other definitions can be used depending on the logic by which properties are expressed.

2.2 Property Specification

The second ingredient of a model checker is a property to check on the system. More precisely, we want to reason on the traces of the system modelled as a TS, and specify which (types of) traces are not accepted. For instance, two properties that are often desired for the system are *"the system is **never** in some failure state"* and *"each time the system receives a request, it **eventually** fulfils that request"*. The properties to verify have thus a *temporal* nature.

Temporal logics are a particular instantiation of modal logics meant to express properties over the lifetime of an entity. They are typically equipped with operators to specify desired behaviour over time. The most common temporal logics are the μ-calculus [124], and in particular two of its fragments: the Computation Tree Logic (CTL) [46] and the Linear Temporal Logic (LTL) [148].[2] We focus particularly on the latter, although we use an extension of the former in Chapter 7.

Definition 2.4 (LTL syntax)
An LTL formula ϕ over a set AP of atomic propositions is formed according to the following grammar:

$$\phi ::= \top \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \bigcirc\phi_1 \mid \phi_1 \mathrm{U} \phi_2$$

where ϕ_1 and ϕ_2 are LTL formulae, $a \in AP$, $\bigcirc$ is the next operator and U *is the until operator.*

An LTL formula is defined over a trace of a given TS, and accordingly specifies discrete-time constraints on this trace. In addition of the usual Boolean connectors and the atomic propositions, it provides two temporal operators. Intuitively, $\bigcirc\phi$ is satisfied at a specific position in the trace if and only if ϕ is satisfied at the next position, that is, after the execution of a transition. The until operator allows one to specify properties over a longer period of time. Basically, $\phi_1 \mathrm{U} \phi_2$ means that ϕ_2 must be eventually satisfied at some point in the future, and that ϕ_1 must be always satisfied before ϕ_2 is satisfied for the first time. Formally, the semantics of LTL is given as follows.

[2] The semantics of TS we gave earlier is not compatible with CTL, and thus nor with the μ-calculus. Nevertheless, this is not an issue since we mostly focus on LTL in what follows.

Definition 2.5 (LTL semantics)
The satisfiability of LTL formulae by a trace σ is defined according to the following rules:

$$\begin{array}{lcl}
\sigma \models \top & \Leftrightarrow & \textit{true} \\
\sigma \models a & \Leftrightarrow & a \in \sigma[0] \\
\sigma \models \phi_1 \wedge \phi_2 & \Leftrightarrow & \sigma \models \phi_1 \wedge \sigma \models \phi_2 \\
\sigma \models \neg\phi & \Leftrightarrow & \sigma \not\models \phi \\
\sigma \models \bigcirc\phi & \Leftrightarrow & \sigma[1\ldots] \models \phi \\
\sigma \models \phi_1 \mathrm{U} \phi_2 & \Leftrightarrow & \exists j \bullet \sigma[j\ldots] \models \phi_2 \wedge \forall i < j \bullet \sigma[i\ldots] \models \phi_1
\end{array}$$

where $\sigma[k]$ denotes the $(k+1)$-th elements of σ, and $\sigma[k\ldots]$ denotes the suffix of σ starting from the $(k+1)$-th element. Then the semantics of a formula ϕ is the infinite set of traces that satisfy ϕ:

$$[\![\phi]\!] = \{\sigma \in (2^{AP})^\omega \mid \sigma \models \phi\}.$$

From the until operator, one may derive two intensively-used operators: $\Diamond$ (eventually) and $\Box$ (always). The first specifies that an LTL formula must hold in some subsequent point of time. On the contrary, the second means that a formula must hold in every subsequent point of time. Formally, we have

$$\begin{array}{lcl}
\sigma \models \Diamond\phi & \Leftrightarrow & \sigma \models \top \mathrm{U} \phi \\
\sigma \models \Box\phi & \Leftrightarrow & \sigma \models \neg(\Diamond\neg\phi).
\end{array}$$

We illustrate these operators in a few examples.

Example 2.2.1 *Let us consider the soda vending machine example. A desired property for this system is that "every ordered soda is eventually served". If we consider that property from the perspective of the state of the machine and assume that the machine can serve one drink at a time, the property consists in "each time the machine is in a state where it receives an order, it will eventually be in a state where it has served the beverage."*

In LTL, this property can be specified using the $\Box$ and the $\Diamond$ operators. Let sodaOrdered be a proposition satisfied when the machine has received a not-yet-delivered order (i.e. it holds in state 5), and sodaServed the proposition satisfied when the machine is in a state where it has just finished serving a soda (i.e. in state 7). The corresponding formula is then $\Box(sodaOrdered \Rightarrow \Diamond sodaServed)$.

2.3 Model-Checking Algorithm

Given a system modelled as a TS named ts and a property expressed as an LTL formula ϕ, checking the system against the property is equivalent to verifying that each trace of the TS satisfies the LTL formula, that is,

$$ts \models \phi \Leftrightarrow \forall \sigma \in [\![ts]\!] \bullet \sigma \models \phi.$$

Fundamentally, an LTL formula ϕ defines a set of accepted traces. For example, the formula $\Diamond a$ can be regarded as the set of traces that have at least one element including a. The satisfaction of ϕ by ts therefore comes down to $[\![ts]\!]$ being a subset of $[\![\phi]\!]$.

Vardi and Wolper [158] have proven that such a verification reduces to checking the non-existence of a trace accepted by both ts and an automaton representing the complement of ϕ, that is, $[\![ts]\!] \cap [\![\mathcal{A}_{\neg\phi}]\!] = \emptyset$. This automaton, whose language is a set of infinite traces, is named a *Büchi automaton.*

Definition 2.6 (Büchi automaton)
[16] A Büchi automaton (BA) is a tuple $ba = (Q, \Sigma, \delta, Q_0, A)$ *where*

- Q *is a set of states,*
- Σ *is the alphabet,*
- $\delta \subseteq Q \times \Sigma \times Q$ *is the transition relation,*
- $Q_0 \subseteq Q$ *is a set of initial states,*
- *and* $A \subseteq Q$ *is a set of accepting states.*

The language of a BA is the set of words leading to a run that visits accepting states infinitely often, that is,

$$[\![ba]\!] = \{\sigma \in \Sigma^\omega \mid \exists q_0 \in Q_0 \bullet \delta^*(q_0, \sigma) \neq \emptyset \wedge \forall k > 0 \bullet \exists j > k \bullet \sigma[j] \in A\}$$

where $\delta^*(s, w)$ *is the set of states that can be reached from* s *by reading the finite word* w*.*

Vardi and Wolper [158] have shown that the intersection between the traces of a TS and the language of a BA is empty if and only if the language of the BA resulting from the synchronous product of the TS and the BA is empty as well.

Definition 2.7 (Synchronous product)
[158, 16] The synchronous product of a TS $ts = (S, Act, Trans, I, AP, L)$ *and a BA* $ba = (Q, 2^{AP}, \delta, Q_0, A)$ *is the BA* $ts \otimes ba = (S \times Q, Act, \delta', I \times Q_0, S \times A)$ *where* $((s,q), \alpha, (s',q')) \in \delta' \Leftrightarrow s \xrightarrow{\alpha} s' \wedge (q, L(s), q') \in \delta$.

The following property summarizes the principles of LTL model checking as described above.

Property 2.1 *[16] Let ts be a TS and* ϕ *be an LTL formula. Then the following holds.*

$$\begin{array}{lcl} & ts & \models \phi \\ \Leftrightarrow & [\![ts]\!] & \subseteq [\![\phi]\!] \\ \Leftrightarrow & [\![ts]\!] \cap [\![\neg\phi]\!] & = \emptyset \\ \Leftrightarrow & [\![ts]\!] \cap [\![\mathcal{A}_{\neg\phi}]\!] & = \emptyset \\ \Leftrightarrow & [\![ts \otimes \mathcal{A}_{\neg\phi}]\!] & = \emptyset. \end{array}$$

Formally, the non-emptiness condition of the language in a BA $(Q, \Sigma, \delta, Q_0, A)$ is given by

$$\exists i \in I \bullet \exists a \in A \bullet \exists (w,v) \in \Sigma^* \times \Sigma^+ \bullet a \in \delta^*(i,w) \cap \delta^*(a,v).$$

If we denote by $R(q,q')$ the fact that state q' can be reached from state q in at least one transition, the above condition can be re-written as $(i = a \vee R(i,a)) \wedge R(a,a)$. We can thus determine if the language of the synchronous product is empty by first computing the set of reachable accepting states and, for each such accepting state a, checking if there exists a non-empty path from a to itself. This can be achieved by performing a nested search, i.e. one search to detect the accepting states and for each discovered accepting states a, one to determine whether a lies in a cycle.

The size of the state space of the Büchi automaton corresponding to an LTL formula is in the worst case exponential in the size of the formula. Moreover, the size of the synchronous product of a TS with a BA is the product of the TS size with the BA size. This leads us to the following complexity result for LTL model checking.

Theorem 2.1
[158, 16] The worst-case time complexity of checking a TS ts against an LTL formula ϕ *is bounded by* $\mathcal{O}(|ts|.2^{|\phi|})$.

The realm of model checking extends itself way beyond the above techniques. If we remain in the scope of systems modelled as a TS, other algorithms are used depending on the logic specifying the properties. For instance, algorithms for CTL consists in decomposing the formula, and then recursively computing the set of states satisfying the obtained subformulae [47]. The fact that the TS and the formula are manipulated directly instead of being transformed into a synchronous product paves the way for symbolic (as opposed to explicit) model checking [141]. This technique consists in encoding the state space and the transition relation as a formula. The verification then comes down to manipulating and checking the satisfiability of Boolean formulae. Symbolic model checking was shown to be more scalable than their explicit counterpart in some cases [33]. Still, the two techniques are to be seen as complementary.

A limitation of TS is that they are only able to express discrete behaviour. Several extensions were proposed to model continuous behaviour. For instance, timed automata [82] were defined to model systems that have to cope with real-time requirements. Markov chains [139] and Markov decision processes [19] are commonly used to represent systems whose behaviour is subjected to random events. Variants of CTL [7, 108] were introduced to specify properties on real-time and stochastic models. Many other extensions exist and have contributed to make model checking a proper discipline. Despite their high interest, covering them entirely would be unreasonable. Therefore, in the subsequent chapters we detail only the theories and techniques that are actual prerequisites for our work.

2.4 Model Checking Versus Program Verification

In addition to model checking, there exist other types of formal software verification techniques. Two of the most common are *static analysis* and *bounded model checking*. The former includes all kinds of techniques for automatically computing information about the behaviour of programs without executing them. However, numerous questions about programs are either undecidable (e.g. the termination of a program) or computationally infeasible to answer. Therefore, the objective of static analysis is only to provide some guarantees about a given program that are not misleading [83]. The principle of these techniques is to propagate values across the program to detect inconsistencies (e.g. division by zero). Since they work directly on source code, they have to deal with various input parameters that can take a potentially wide range of values. In order to avoid performing the verification for all combinations of parameter values, they typically rely on abstract interpretation [71]. This consists in evaluating the behaviour of the program based on an abstract representation of sets of concrete values in order to obtain an approximate but sound answer to a given problem. Static analysis

has the advantage of being sufficiently scalable to work on large programs with varied types of input. Yet, the range of properties they can check is limited; for instance they cannot verify all properties expressible in LTL. Moreover, these verifications are generally very specific and hard coded into the static analyser. A user has thus not the freedom to specify additional properties she wants to check on her program. Finally, it is difficult to generate a counterexample to illustrate a detected error [83].

Bounded model checking can be applied on design models or on the software itself. In the latter case, this technique first converts the basic blocks of the program and the transition relation between these blocks into propositional formulae. States of the program are also represented as a predicate. Then, by iteratively applying the formulae obtained from the basic blocks and the transition relation on the predicate of the initial state, the checker obtains another formula representing the state of the program after a certain number k of executed transitions. This formula can then be conjoined with yet another formula representing the negation of a property to check. Finally, the resulting formula is fed into a satisfiability solver that determines whether the formula is satisfiable, or equivalently if there exists a trace prefix of length k that violates the property. Bounded model checking is inherently incomplete, as it searches through the state space of the program up to a given bound. Still, it is recognized as one of the best techniques to find shallow bugs [83]. It is also able to provide a counterexample in case of violation, and it supports a large number of program constructs. Its incompleteness can be somehow alleviated by setting a *sufficient* bound, but determining an appropriate threshold is as hard as the model-checking problem itself [83].

In spite of their seeming differences with model checking, it is noteworthy that both static analysis and bounded model checking also consist in exploring a graph, although this graph is richer than the TS classically checked by a model checker. Extending the two techniques to make them coping with variability would require the development of heuristics highly similar to the ones we present in the next chapters. This means that the methods we propose in this thesis, or at least their principles, are also applicable to program verification in addition to model checking.

Chapter 3

Featured Transition Systems

Model checking as presented in the previous chapter is intended to verify single systems. When considering SPLs, the model-checking problem takes a different form: it requires to verify all the products that can be built against a given property (in our case, expressed in LTL). More precisely, it is desired to identify exactly which SPL products violate the property [56].

Definition 3.1 (SPL model checking)
Let fm be a feature model, M_v be a behavioural model of all products in $[\![fm]\!]$, and ϕ a property. Model checking M_v against ϕ is the problem of:

1. *Determining for each product $p \in [\![fm]\!]$ whether p satisfies ϕ in M_v, that is, whether the behaviour of p expressed in M_v satisfies ϕ.*
2. *Providing for each product p that does not satisfy ϕ a counterexample of behaviour of p that violates ϕ.*

From the theory presented in Chapter 2, one can already derive a method to address the SPL model-checking problem. This method consists in modelling every valid product of an SPL in a separate TS, and then applying Vardi and Wolper's algorithm on each of these TS individually. This method, named *enumerative* [58] or product-based [159], immediately appears against the principles of SPL engineering: the products should not be modelled separately. Instead, one should build a core model, which is subsequently derived into desired variants. In addition to the modelling task, performance is also a major concern. State explosion, a problem inherent to model checking, is amplified when considering SPLs, especially when these consist of a huge number of products. Being part of an SPL, these products likely share commonalities in both their structure and their behaviour.

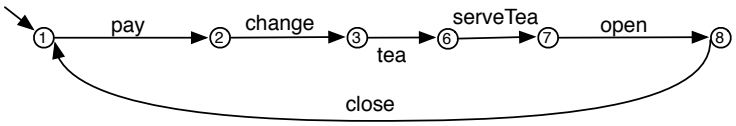


Figure 3.1: A TS modelling a tea vending machine

Consider, e.g., an alternative variant of the vending machine that serves tea instead of soda. A TS modelling its behaviour is shown in Figure 3.1. The sole differences between this TS and that of Figure 2.1 are the transition from state 3 to state 6 and the one from state 6 to state 7. Basically, these transitions are responsible for receiving an order for tea and delivering the beverage, respectively. The five other transitions are common to the two products. Hence, the verification of a property whose satisfaction does not depend on the type of delivered drinks would yield the same results for the two TS. This observation illustrates the fact that single-system verification techniques are suboptimal to address the SPL model-checking problem. Clearly, SPL model checking would benefit from models that can concisely represent the behaviour of a set of products, and algorithms that can exploit the information about the commonalities between these products to facilitate verification.

3.1 A Formalism to model SPL Behaviour

Classen *et al.* [59, 58] proposed Featured Transition Systems (FTS) as a compact representation for a set of products. These are an extension of TS equipped with an FM and whose every transition is annotated with the exact set of products able to execute it. For the sake of conciseness, these sets are encoded as feature expressions (see Definition 1.6).

Definition 3.2 (Featured transition systems)
[58, 56] *An FTS is a tuple* $(S, Act, Trans, I, AP, L, fm, \gamma)$*, where*

- $S, Act, Trans, I, AP, L$ *are defined as in Definition 2.2;*
- fm *is a FM over a set of features* F*;*
- $\gamma : Trans \to 2^{(2^F)}$ *is a total function that associates a transition with a feature expression over* F*.*

An FTS can be seen as the merging of the TS of all the products that

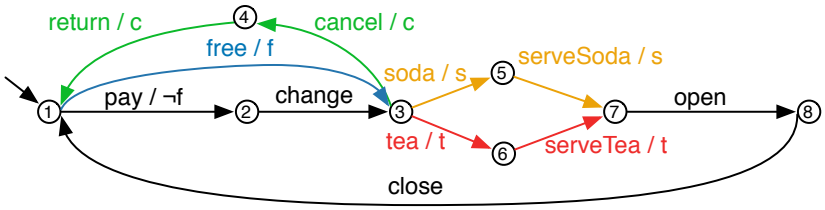


Figure 3.2: The FTS modelling the vending machine SPL.

compose the SPL. The TS modelling a specific product is obtained from the FTS by applying a *projection* function. In simple terms, this function prunes the FTS of all the transitions whose feature expression is not satisfied by the considered product [59], and then removes all feature expressions.

Definition 3.3 (Projection of FTS)
[56] Let $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ *be an FTS and* $p \in [\![fm]\!]$ *be a product. The projection of* fts *onto* p*, noted* $fts_{|p}$*, is the TS* $(S, Act, Trans', I, AP, L)$ *where*

$$Trans' = \{t \in Trans \mid p \models \gamma(t)\}.$$

Example 3.1.1 *Figure 3.2 depicts an FTS modelling an SPL of vending machines, while Figure 3.3 shows its associated FM. This SPL consists of 12 products, each of which has its behaviour modelled by the FTS. For instance, the transition from state 3 to state 6 is labelled with the feature expression* t*, meaning that it can be executed only by products including the corresponding feature* Tea*. Transition from state 1 to state 2 is labelled with* $\neg f$*, and thus can be executed only by products that do not have the feature* $Free$*.*

Since an FTS represents the behaviour of a *set* of products, its semantics is defined as a function that associates a product with the traces of the corresponding projection.

Definition 3.4 (FTS Semantics)
Let fts *be an FTS over an FM* fm*. The semantics of* fts *is a total function* $[\![fts]\!] : [\![fm]\!] \to 2^{(2^{AP})^\omega}$ *such that*

$$\forall p \in [\![fm]\!] \bullet [\![fts]\!](p) = [\![fts_{|p}]\!].$$

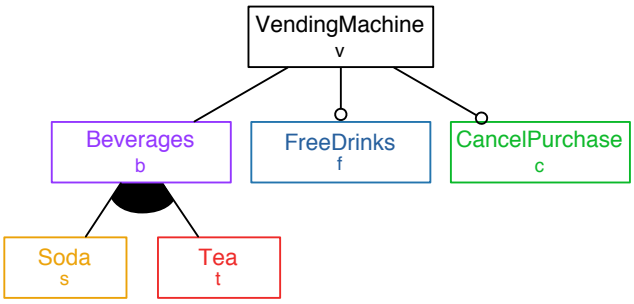


Figure 3.3: The FM of the vending machine SPL.

Note that Classen *et al.* gave a different semantics to FTS. They defined it as the union of the semantics of all projections [59], i.e. $\cup_{p \in [\![d]\!]} [\![fts_{|p}]\!]$. We argue that this alternative definition is not appropriate for two reasons. First, it does not express any traceability link between products and behaviour, and thus theoretically makes it impossible to identify the products violating a given property. Second, which is a consequence of the first, a product whose set of traces is subsumed by the set of traces of another product disappears from the semantics. Paradoxically, the algorithms designed by Classen *et al.* [59, 58, 56] (see further in Section 3.3) assume the existence of the traceability link, and are thus implicitly based on Definition 3.4. Our definition reflects thus more accurately Classen *et al.*'s work and we stick to it in the rest of this manuscript.

Another interesting observation is that if we consider an FTS as a TS (that is, we replace every feature expression by a tautology), the semantics of that TS would subsume the semantics of any individual projection. This is because traces including transitions with incompatible feature expressions (i.e. that encode disjoint sets of products) are part of the TS although no valid product can actually execute them. We can thus regard this TS as a coarse abstraction of the FTS. We elaborate this discussion in Chapter 6, where we present abstraction methods.

3.2 FTS Model Checking

Contrary to single systems, a binary result is not sufficient to appropriately address the model-checking problem for SPLs. In case of property violation, a model checker is expected to identify all products responsible for the violation. There is thus a need for generalizing the definition of model checking. We already gave it an intuitive definition at the beginning of this chapter. Here, we rephrase this definition formally, by considering FTS as a model for SPL behaviour.

Beforehand, let us remark that a property may only be relevant for

certain products. For instance, as observed by Classen *et al.* [59], a property may refer to characteristics that only occur in a subset of the SPL. To address this requirement, they proposed to extend temporal logics with a *product quantifier*, i.e. a feature expression that defines for which products the property must be checked [59, 58]. The resulting variant of LTL is defined as follows.

Definition 3.5 (fLTL)
[59, 56] Let F be a set of features. An fLTL formula ψ is an expression $\psi = [\chi]\phi$ where χ is a feature expression over F and ϕ an LTL formula. Let fts be an FTS over an FM fm over F and let $p \in [\![fm]\!]$. Then p satisfies ψ in fts if and only if $\chi(p) \Rightarrow fts_{|p} \models \phi$.

We are now ready to define a new notion of satisfiability specific to product lines.

Definition 3.6 (F-satisfiability)
Let fts be a FTS over an FM fm, and $\psi = [\chi]\phi$ be an fLTL formula. Then, the products that F-satisfy ψ in fts are encoded as the feature expression

$$(fts \models \psi) = \neg\chi \vee \mathbb{B}(\{p \in [\![fm]\!] \mid fts_{|p} \models \phi\}).$$

Conversely, the products that F-unsatisfy ψ in fts are encoded as the feature expression

$$(fts \not\models \psi) = \chi \wedge \mathbb{B}(\{p \in [\![fm]\!] \mid fts_{|p} \not\models \phi\}).$$

Given an FTS and an fLTL formula, an SPL model checker should thus compute the corresponding F-satisfiability, and associate each F-unsatisfying product with one of its traces that violates the formula. Note that according to the above definition, $\{[\![fts \models \psi]\!], [\![fts \not\models \psi]\!]\}$ is a partition of $[\![fm]\!]$.

3.3 Algorithms

Given its explicit notion of features, FTS constitutes a suitable formalism to concisely model behaviour subject to variability. Yet there remains a second challenge to solve, i.e. an efficient verification of the behaviour of a set of products. To achieve that, Classen *et al.* designed algorithms to check FTS

against LTL [59, 56] and CTL [58] formulae that exploit common transitions among products to reduce the verification effort. As opposed to the product-based approach, a given behaviour is not always checked as many times as the number of products in which it occurs.

Let us consider properties expressed as fLTL formulae. As mentioned in Section 2.3, verifying a TS against such properties comes down to computing reachability relations in order to determine the existence of an accepting cycle in a BA. Classen *et al.* apply the same principles for FTS and propose to compute the synchronous product of an FTS with a BA. The resulting automaton has to maintain the variability encoding contained in the FTS so that exploration algorithms can still make use of this encoding. This gives rise to a new formalism that includes both an infinite language and variability information.[1]

Definition 3.7 (Featured Büchi automaton)
A Featured Büchi Automaton (FBA) is a tuple $(Q, \Sigma, \delta, Q_0, A, fm, \gamma)$ *where* Q, Σ, δ, Q_0, *and* A *are as in Definition 2.6,* fm *is an FM over a set of features* F, *and* $\gamma : \delta \to 2^{(2^F)}$ *has the same role as in Definition 3.2.*

Similarly to FTS, the semantics of an FBA is a function associating a product p with the language accepted by the projection of the automaton onto p. The definition of projection for FBA follows the same pattern as that of FTS and is thus not given here.

The synchronous product of an FTS and a BA gives rise to an FBA defined similarly to the synchronous product of a TS with a BA and whose transition labelling function is derived from that of the FTS.

Definition 3.8 (Synchronous product of an FTS and a BA)
The synchronous product of an FTS $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ *and a BA* $ba = (Q, 2^{AP}, \delta, Q_0, A)$ *is the FBA* $fts \otimes ba = (S \times Q, Act, \delta', I \times Q_0, S \times A, fm, \gamma')$ *where* δ' *is defined as in Definition 2.7 and for any tuple* $(s, q, \alpha, s', q') \in \delta'$ *we have* $\gamma'(s, q, \alpha, s', q') = \gamma(s, \alpha, s')$*.[59]*

As for TS model checking, the algorithm of Classen *et al.* for FTS [59, 56] checks the emptiness of the language of the synchronous product. The dif-

[1] This type of automaton has not been explicitly defined by Classen *et al.*, who define the synchronous product of an FTS with a BA as an FTS with an altered semantics [56]. In order to remain consistent with the vocabulary and notations of single-system model checking, we took the freedom to introduce a proper definition.

ference is that it performs this verification for all SPL products. The correctness of this algorithm relies on the property that projection is distributive over the synchronous product operator modulo semantic equivalence.[2]

Theorem 3.1
Let fts be an FTS over an FM fm, and ba be a BA. Then for any $p \in [\![fm]\!]$ it holds that

$$[\![(fts \otimes ba)_{|p}]\!] = [\![fts_{|p} \otimes ba]\!].$$

As in its single-system counterpart, the algorithm first looks for accepting states in the FBA starting from an initial state, and then determines whether one of them can reach itself via a non-empty path. Both steps come down to a reachability computation. A major difference is that reachability now depends on variability: for a given initial state s_0 and an accepting state a, the products that can visit a infinitely often starting from i are those that can reach a from i and from a itself. More generally, the products for which the language of the FBA is non-empty are obtained by existentially quantifying the above condition over all initial states and accepting states. To identify these products, we can perform a nested search (as in Vardi and Wolper's algorithm) while keeping track of the products able to execute a given path explored by the algorithm. Fundamentally, the difference between the two algorithms is the definition of successor state. In TS (resp. BA), state s' is a successor of a given state s if and only if there exists a transition from s to s'. In FTS (resp. FBA), variability can influence the set of successors as a transition may exist for only a subset of the products. The definition of successor has thus to be revisited according to products as well. It is given as follows.

Definition 3.9 (Successors in FTS)
Let $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ be an FTS. The successor function in fts is defined as $Post : S \to (S \to 2^{(2^F)})$ such that:

$$\begin{aligned} Post(s_i)(s_{i+1}) &= \mathbb{B}(\{p \in \cup_\alpha [\![\gamma(s_i, \alpha, s_{i+1})]\!]\}) \\ &= \bigvee_\alpha \gamma(s_i, \alpha, s_{i+1}) \end{aligned}$$

with $(s_i, \alpha, s_{i+1}) \in Trans$.

[2]The term *distributive* is not really appropriate here since projection of BA is not defined. Still it is consistent to define that projection has no effect on BA. Under this consideration, the use of the term is correct.

Intuitively, for a given couple of states (s, s') the function $Post(s)(s')$ is the feature expression encoding the products that can execute a transition from s to s'. The definition of successors in FBA is very similar and not given here.

From the definition of successor, one can define reachability relation in FTS. Similarly to successor, reachability takes the form of a function. It associates two states, say s_0 and s_n, to a feature expression encoding the products able to reach s_n from s_0. These products are those able to follow at least one path from s_0 to s_n. Let $s_0, \ldots, s_n$ be a path in a given FTS. A product can follow this path if and only if it satisfies the feature expression $\bigwedge_{0 \leq i < n} Post(s_i)(s_{i+1})$. To obtain the products that can reach s_n from s_0, we can existentially quantify the above expression over all the paths from s_0 to s_n.

Definition 3.10 (Reachability in FTS)
Let $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ be an FTS. Reachability in fts is a function $R : S \to (S \to 2^{(2^F)})$ such that:

$$
\begin{aligned}
R(s_0)(s_n) &= \mathbb{B}(\{p \in 2^F \mid \exists s_1, \ldots, s_{n-1} \bullet p \in [\![\bigwedge_{i=0}^{n-1} Post(s_i)(s_{i+1})]\!]\}) \\
&= \mathbb{B}(\{p \in [\![\bigvee_{s_1,\ldots,s_{n-1} \in S^{n-1}} \bigwedge_{i=0}^{n-1} Post(s_i)(s_{i+1})]\!]\}) \\
&= \bigvee_{s_1,\ldots,s_{n-1}} \bigwedge_{i=0}^{n} Post(s_i)(s_{i+1})
\end{aligned}
$$

where $\forall j \bullet 0 \leq j < n \bullet \exists \alpha \in Act \bullet (s_j, \alpha, s_{j+1}) \in Trans$.

Once again, the definition of reachability function in FBA is analogous and therefore omitted.

Finally, the above definition of reachability allows us to characterize the products that satisfy an LTL formula in a given FTS.

Theorem 3.2
Let fts be an FTS, and ϕ be an LTL formula. Then the feature expression encoding these products, noted $fts \models \phi$, is given by:

$$\bigvee_{(q_0 \in Q_0, a \in A)} R(q_0, a) \wedge R(a, a)$$

where Q_0 and A are respectively the set of initial states and the set of accepting states in $fts \otimes \mathcal{A}_{\neg\phi}$.

To efficiently compute the reachability function in an FTS or an FBA, Classen *et al.* designed a nested depth-first search algorithm that accumulates the conjunction of the feature expressions of all transitions executed on a given path in order to keep track of the products able to reach any state met along this path. This means that the algorithm separates the verification of different sets of products only if they discover a behavioural discrepancy between them. This optimisation is called *late splitting* [13].

Algorithm 1 formalises this process to compute the reachability function of a given state s_0. The algorithm consists of a loop that iterates over a stack of couples (s, γ) where s is a state and γ is a feature expression. Initially, the stack contains only the element (s_0, fm) in order to start the search from s_0 while considering all the products. At each iteration, the algorithm takes the top element (s, γ) of the stack, computes the successors of s and associate each successor with the products that satisfy γ and can reach the successor from s (Lines 4–5). This results in a set of couples $(s', \gamma') \in S \times 2^{(2^F)}$. For each such couple, the algorithm first determines whether $[\![\gamma']\!]$ contains at least one valid product; otherwise it is not needed to pursue the search from s'. This verification is achieved by checking the satisfiability of γ' (Line 6). If that is the case, we enter an inner loop (Lines 7–17).

During the search, the algorithm may visit a given state more than once (Lines 7–13). In single-system model checking, it should not pursue the search since it already knows that the revisited state is reachable. In our case, however, it may happen that the algorithm discovers a new path to an already visited state s' which is executable by products that were not known to be able to reach s'. Formally, let $R(s')$ be the feature expression encoding the set of products that were known to reach s'. Then $\neg R(s') \wedge \gamma'$ encodes the set of products that are newly known to reach s (noted γ_{new} at Line 8). If there is at least one valid product satisfying this feature expression, the search continues from s' considering only the products in γ_{new} (Lines 9–12). Indeed, any state reachable from s for products $[\![R(s')]\!]$ may have already been visited for these products. Therefore, the paths starting from s are worth re-exploring only for the products in γ_{new}. Before pursuing the exploration, the feature expression $R(s')$ is updated accordingly.

Given the above algorithm, the theoretical complexity of model checking FTS against LTL formulae is given as follows.

Input: $fts = (S, Act, Trans, I, AP, L, fm, \gamma), s_0 \in S$.
Output: $R(s_0)$.

```
1  R ← ⊥;
2  Stack ← push((s_0, fm), []);
3  while Stack ≠ [] do
4  |  (s, γ) ← pop(Stack);
5  |  succ ← {(s', γ') | s' ∈ dom(Post(s)) ∧ γ' = Post(s)(s') ∧ γ};
6  |  foreach (s', γ') ∈ succ • γ' ⊭ ⊥ do
7  |  |  if s' ∈ dom(R) then
8  |  |  |  γ_new ← ¬R(s') ∧ γ';
9  |  |  |  if γ_new ⊭ ⊥ then
10 |  |  |  |  R(s') ← R(s') ∨ γ';
11 |  |  |  |  push((s', γ_new), Stack);
12 |  |  |  end
13 |  |  end
14 |  |  else
15 |  |  |  R(s') ← γ';
16 |  |  |  push((s', γ'), Stack);
17 |  |  end
18 |  end
19 end
20 return R
```

Algorithm 1: Reachables(fts, s_0)

Theorem 3.3
[56] Let fts be an FTS over a set of features F and ϕ be an LTL formula. The worst-case time complexity of computing $fts \models \phi$ is bounded by $\mathcal{O}(|fts|.2^{|\phi|}.2^{2.|F|})$.

Intuitively, in the worst-case each valid product has a different behaviour starting from the initial state. In this case, Classen *et al.*'s algorithm behaves as an enumerative application of Vardi and Wolper's algorithm on every such product. Moreover, the number of valid products is in the worst-case the size of the power set of F, i.e. $2^{|F|}$. Furthermore, there is an overhead in the FTS algorithm that does not exist in the product-by-product method: At each iteration, a satisfiability check on feature expression is performed, which also has a time complexity of $\mathcal{O}(2^{|F|})$. Although the FTS algorithm has a worse theoretical complexity, experiments tend to show that in practice it outperforms the product-based approach [59, 58, 53, 56]. The FTS theory is thus a solid candidate solution for the SPL model-checking problem.

Chapter 4

Alternative Approaches

FTS are certainly neither the only nor the first approach for SPL verification. This chapter aims at presenting the concurrent approaches, focusing particularly on two of the most developed, *viz.* modal transition systems and multi-valued models. We successively present these two and compare them with FTS along two axes: expressiveness of the formalism and optimisation in the algorithms. Afterwards, we give and overview of other related work.

4.1 Modal Transition Systems

Fischbein *et al.* [90] were the first to propose Modal Transition Systems (MTS) [116] as a foundation for modelling SPL behaviour. MTS are an extension of TS where transitions are either *mandatory* or *optional*.

Definition 4.1 (Modal transition system)
An MTS is a tuple $(S, Act, May, Must, I, AP, L)$ *where*

- S, Act, I, AP, *and* L *are defined as in Definition 2.2;*
- $May \subseteq S \times Act \times S$ *is the possible transition relation;*
- $Must \subseteq May$ *is the mandatory transition relation.*

Accordingly, the optional transition relation is given by $May \setminus Must$.

From an SPL perspective, mandatory transitions are transitions available to all valid products, whereas optional transitions model those triggered by features owned only by a strict subset of the valid products. Similarly, we can categorize traces of an MTS into mandatory and optional traces.

Let fm be an FM and mts (respectively fts) be an MTS (respectively an FTS) modelling the behaviour of the products in $[\![fm]\!]$. We can define the semantics of mts in terms of FTS as $[\![mts]\!] = [\![mts_{|must}]\!] \cup [\![mts_{|opt}]\!]$ where

- $[\![mts_{|must}]\!] = \{\sigma \in (2^{AP})^\omega \mid \forall p \in [\![fm]\!] \bullet \sigma \in [\![fts]\!](p)\}$;
- $[\![mts_{|opt}]\!] = \{\sigma \in (2^{AP})^\omega \mid \exists p \in [\![fm]\!] \bullet \sigma \in [\![fts]\!](p)\} \setminus [\![mts_{|must}]\!]$.

This implies that for any FTS we can define an MTS which is equivalent in the sense of the above semantics. However, transforming this FTS back into the original FTS is impossible as the latter holds information that the former does not. Indeed, in an MTS it is impossible to relate a given transition to the exact set of products able to execute it. FTS is thus *strictly more expressive* than MTS. To make MTS overcome this limitation, Asirelli *et al.* [15] associated the formalism with a branching-time temporal logic named MHML. Formulae in this logic allows one to specify constraints over actions that may or will be executed in the future. Through an association between actions of optional transitions and features, it is thus possible to define MHML formulae that represent constraints between the features, thereby forbidding executions that require invalid combinations of features. The TS modelling the individual products are obtained by a synthesis process driven by the MHML formulae [15]. Another benefit of associating actions to features lies in the derived capability of linking a trace to the products able to execute it. This makes the combination of MTS and MHML as expressive as FTS, and also enables the use of efficient algorithms similar to those presented in Section 3.3.

However, the need for an additional logic raises a number of issues with regard to modelling facilities and tool support. SPL variability is traditionally modelled as an FM. The approach of Asirelli *et al.* is based on an a priori encoding of all the constraints expressed by the FM into MHML. This encoding cannot be realized from the FM alone, as it requires an explicit link to the actions of the MTS modelling the behaviour of the products. This link is also needed to apply FTS-based algorithms, since these accumulate feature expressions as the exploration goes on. MHML thus adds a unneeded level of indirection, which accidentally complicates the modelling tasks.

4.2 Multi-Valued Model Checking

Multi-valued model checking [39, 38, 106, 31] generalises the classical model-checking problem. In multi-valued models, the transition relation is not binary and atomic propositions are not Boolean. Instead, their values are defined over an arbitrary value lattice.

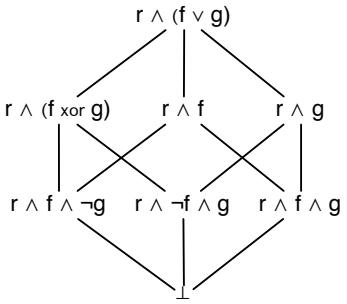


Figure 4.1: A lattice representing the valid products of an FM.

Definition 4.2 (Lattice)
A lattice is a partial order $(\mathcal{L}, \sqsubseteq)$ *such that for any couple* $(a, b) \in \mathcal{L} \times \mathcal{L}$*, there exist a unique greatest lower bound* $a \sqcap b$ *and a unique least upper bound* $a \sqcup b$*.*

The top element of a lattice, if it exists, is generally noted $\top$, whereas the bottom one is always $\bot$. Elements at the level just above $\bot$ are named *join irreducible*. In our context, elements of the lattice are sets of products, which we assume to be encoded as feature expressions. Hence, $\sqcap$ and $\sqcup$ are respectively the conjunction and disjunction of Boolean formulae. The join-irreducible elements then correspond to singletons of a single product, or equivalently a feature expression that consists of a conjunction of literals where any feature occurs in either positive or negative form. For example, let us consider an FM over features $\{r, f, g\}$ where r is the root with $children(r) = \{f, g\}$ and $\lambda(r) = (1, 2)$. The lattice representing all possible products is the set of Boolean formulae over this set of features. However, we are interested only in the valid products, i.e. $\{r, f\}$, $\{r, g\}$, and $\{r, f, g\}$. The lattice should thus be composed of feature expressions encoding these three sets. Figure 4.1 shows this lattice. As previously mentioned, the join-irreducible elements encode the three valid products. The elements of the level just above encode pairs of valid products. Finally, the top elements encode the set containing the three products. It is also the feature expression equivalent to the FM.

The concept of lattice leads us to the definition of multi-valued transition systems.[1]

[1]This definition slightly differs from Chechik *et al.* [40] in that it generalizes transition systems, whereas Chechik *et al.* consider Kripke structures.

Definition 4.3 (Multi-valued transition system)
A multi-valued transition system is a tuple $(S, Act, Trans, I, AP, L, \gamma, (\mathcal{L}, \sqsubseteq))$ *where*

- $S, Act, Trans, I$, *and* AP *are defined as in Definition 2.2;*
- $(\mathcal{L}, \sqsubseteq)$ *is a value lattice;*
- $\gamma : Trans \to \mathcal{L}$ *is a total function that associates each transition to a value in the lattice;*
- $L : S \times AP \to \mathcal{L}$ *is a total function that associates a state and an atomic proposition to a value in the lattice.*

The γ function in the above definition has the same role as in FTS, that is, to restrict transitions to certain elements of the lattice. FTS can thus be seen as a particular case of multi-valued TS, where L returns Boolean values and the transition relation is defined over a lattice whose values are feature expressions. In contrast to the generic multi-valued algorithms, the FTS algorithm exploits these particularities to improve efficiency. Existing multi-valued model checking algorithms for LTL are based on the projection of a model onto each irreducible element of its value lattice. In the context of FTS, this is equivalent to the enumerative algorithm, which checks each product individually. Benchmarks tend to show that the FTS algorithm outperforms such a method [56]. Chechik *et al.* [38] report on improvements over the projection-based algorithms. However, their proposal still requires to define and manipulate explicitly every element of the value lattice, whose size can be $O(2^{|F|})$. Another advantage of the FTS algorithms over multi-valued algorithms is their ability to use the information contained in the FM, the product quantifier in fLTL formulae, as well as optimisations such as those discussed in [56] to reduce the number of explored states. Moreover, unlike multi-valued models, FTS are specifically designed in the context of SPLs. In addition to the algorithmic improvements, FTS thus come with a frame of reference for SPL engineering. This makes us argue that in spite of a higher expressiveness – which has no use in our context – multi-valued models are less appropriate than FTS.

4.3 Other Related Approaches

As an alternative to MTS, Larsen *et al.* [130] propose modal I/O automata as a formalism to model SPL behaviour. This kind of automata suffers from the same limitations as MTS, and thus requires an additional logic to be

as expressive as FTS. Lauenroth *et al.* [131] define an extension of modal I/O automata that partially overcomes these limitations. Therein, a single feature can be attached to a transition. Although this is already a major improvement over a feature-unaware formalism like MTS, it is still less expressive than FTS, in which arbitrary feature expressions can annotate transitions. Based on their modelling language, they propose an explicit-state CTL model-checking algorithm to determine whether all products satisfy a given property. Although it tackles a simpler problem, this algorithm is much less efficient than the FTS-based ones, as it basically consists in exploring every possible path of the state space.

Another approach is PL-CCS [103]. It extends CCS [145] with a product-line variant operator that models an alternative choice between two processes. The semantics of the language is defined in terms of a multi-valued model. The authors concede that their formalism does not incorporate constraints between features as encoded in an FM. However, they argue that this is indeed possible. Finally, Millo *et al.* [143] define finite state machines with variability annotations. This formalism is actually equivalent to featured program graphs [53], which in turn can be transformed into FTS. The verification procedure proposed by the authors, however, is product-based and thus does not exploit the commonalities between the products.

All the above approaches are *annotative*, i.e. they consist of annotating an existing formalism with additional information. By contrast, there exist *compositional* methods that advocate a separate representation of each feature's behaviour. Fisler and Krishnamurthi [91] propose an approach to determine if a feature added to a base system preserves previously satisfied CTL formulae. They model a feature as a partial TS that connects to the TS modelling the base system. Their method is modular in the sense that only the state space introduced by the feature is visited. However, it is sound but not complete, and it considers only features that strictly add behaviour to the system. Furthermore, both the base system and the feature are modelled as finite state machines. Therefore, if the approach is applied to more than one feature it cannot keep track of all the products; it thus cannot answer the SPL model-checking problem (see Definition 3.1). Li *et al.* [133, 132, 134] extend the above approach and consider features in open systems. Also, they allow the formula expressing a given requirement to evolve after the addition of a feature. Krishnamurthi *et al.* [128, 127] apply this methodology to aspect-oriented programming. More precisely, they present the foundations for a modular and incremental verification of aspects. These approaches also suffer from the above limitations and are defined only for CTL formulae. Liu *et al.* [135] introduce an alternate method to model check an SPL compositionally. From a behavioural model of a system, they derive a set of constraints which must be satisfied by any feature added to this system. Their approach is compositional: not only it determines if the new feature preserves a given property, but it refines the set

of constraints in order to verify subsequent features. However, they do not explicitly keep track of the features and hence of the products. Therefore, it may not always be possible to identify exactly which products do not preserve a property. Unlike the previous approaches, Guelev *et al.* [104] tackle the preservation of LTL formulae and give criteria to deduce that a feature added does not break previously satisfied formulae. While considering more general features, their approach can be applied only to safety properties and particular liveness properties. Another compositional approach is SPLVerifier by Apel *et al.* [12]. Features are modelled as separate and composable units. Each feature includes a safety property and a product is valid if and only if it satisfies the safety properties of its features. Verification is preceded by a *variability encoding* step, which transforms features into variables that are non-deterministically initialised. From there on, SPLVerifier uses a classical model checker to detect violations. This approach cannot identify all products that violate the property since the model checker stops after the first violating product is found. This also makes it hard to determine the features responsible for a violation.

Among this plethora of formalisms lie serious contenders for replacing FTS. A first essential question is whether one should adopt the annotative or compositional paradigm. A definitive answer is difficult to provide. From a pure modelling perspective, compositional solutions appear at first more elegant and scalable than annotative ones since they allow separation of concerns. However, this becomes more complicated as soon as feature interaction occurs. When two features affect each other's behaviour, formal rules have to be given in order to determine how the composition of these features ultimately behaves. These rules can consist of either predefined priority between features [58] or composition rules over the modelling formalism [155]. On the contrary, these ambiguities do not occur in annotative models. More generally, one can transform any compositional model into an annotative equivalent. This means that any theoretical development designed for annotative models can also be reused in compositional approaches. Among the annotative formalisms, we mainly focused on FTS, MTS, and multi-valued TS. The inherent limitations of MTS, although offset by the addition of the MHML logic, made us discard this solution. Multi-valued TS are more expressive than FTS, but lack of efficient algorithms. Moreover, the link between this kind of automaton and essential SPL concepts like features and FM is not immediate. FTS own a sufficient level of expressiveness, are directly defined within the SPL engineering framework, and come together with an already interesting set of optimisations. For these reasons, this thesis relies on this formalism and its previous related developments.

Part II

Abstraction-based Product-Line Verification

Chapter 5

Variability-Aware Simulation Relation

Early experiments have shown that FTS model-checking approaches are more efficient than product-based approaches [59, 58, 56]. Indeed, when comparing the two approaches implemented within tools such as SNIP [55] and fNuSMV [54], we observed that FTS algorithms generally outperform the product-by-product method [59]. However, our experiments also have shown that there is still much room for improvement. SPL verification theory and practice are still at an early stage and need to be further improved to target industry-scale SPL verification.

Model abstraction is an optimisation that that consists in simplifying a model prior to its verification [30]. Roughly speaking, an abstraction function is used to reduce the size of a model by merging similar states. Depending on the definition of this function, the behaviour of the model may change, that is, new behaviours may appear, existing ones may disappear, or both. Two different TS can thus model a software product at different abstraction levels. If a more abstract (that is, smaller) model preserves the properties of a larger model, it is more efficient to check properties on the former. It is therefore essential to be able to relate two models at different abstraction levels. For single systems, this information is formally captured by a *simulation relation* [144].

Definition 5.1 (Simulation relation)
[16] Let $ts_i = (S_i, Act_i, Trans_i, I_i, AP, L_i), i \in \{1, 2\}$ *be TS over AP. A simulation for* (ts_1, ts_2) *is a binary relation* $\mathcal{R} \subseteq S_1 \times S_2$ *such that*

1. $\forall s_1 \in I_1 \bullet \exists s_2 \in I_2 \bullet (s_1, s_2) \in \mathcal{R}$ *and*
2. $\forall (s_1, s_2) \in \mathcal{R}$ *it holds that*

(a) $L_1(s_1) = L_2(s_2)$ *and*

(b) $\forall s_1' \in Post_1(s_1) \bullet \exists s_2' \in Post_2(s_2) \bullet (s_1', s_2') \in \mathcal{R}$.

where $Post_i(s) = \{s' \in S_i \mid \exists \alpha \in Act_i \bullet (s, \alpha, s') \in Trans_i\}$ *denotes the set of states that can be reached from* s *in one transition in* ts_i*. Then,* ts_2 *simulates* ts_1*, noted* $ts_1 \preccurlyeq_{TS} ts_2$ *if and only if there exists a simulation for* (ts_1, ts_2)*.*

According to this definition, if ts_1 is simulated by ts_2, then any trace of ts_1 is also a trace of ts_2 [16]. Also, we say that state s_1 is simulated by state s_2 if and only if $(s_1, s_2) \in \mathcal{R}$ for some $\mathcal{R}$, also noted $s_1 \preccurlyeq_{TS} s_2$. Intuitively, this implies that any trace produced from s_1 can be produced from s_2. Note that $\preccurlyeq_{TS}$ is a preorder – it is reflexive and transitive [16]. Additionally, when $ts_1 \preccurlyeq_{TS} ts_2$ and $ts_2 \preccurlyeq_{TS} ts_1$ hold, the TS are called *simulation-equivalent*, noted $ts_1 \simeq_{TS} ts_2$, which implies that ts_1 and ts_2 have the same traces. Since $\preccurlyeq_{TS}$ is a preorder, $\simeq_{TS}$ is an equivalence relation [16].

The definition of simulation allows one to characterise the behaviour of an abstract TS $\widehat{ts}$ with regard to an original model ts. Informally, an abstract TS is obtained by merging states for which a so-called abstraction function returns the same value (see Chapter 6 for a formal definition of abstraction function). The abstraction may add or remove behavioural options, depending on the chosen abstraction function. However, a relevant analysis requires to have either $ts \preccurlyeq_{TS} \widehat{ts}$, $\widehat{ts} \preccurlyeq_{TS} ts$, or both. If this condition is satisfied, we can show that the abstraction preserves the (un)satisfiability of properties expressed in a given logic.

Indeed, there is a strong link between simulation relations and the properties satisfied by two TS. As mentioned before, we focus on properties expressed in LTL. However, the presented results can also be applied to the universal fragment of CTL. If a simulation relation exists between two transition systems, we can show that an LTL formula satisfied by the simulating TS is preserved in the simulated one.

Property 5.1 *[144, 16] Let* ts_1 *and* ts_2 *be two TS and* ϕ *be an LTL formula. Then*

$$ts_1 \preccurlyeq_{TS} ts_2 \Rightarrow (ts_2 \models \phi \Rightarrow ts_1 \models \phi).$$

The following statement is equivalent: if ts_1 does not satify ϕ, neither does ts_2. Finally, if $ts_1 \simeq_{TS} ts_2$, then they satisfy exactly the same LTL

formulae. In particular, if ts_2 is an abstraction of ts_1, proving that the abstract TS satisfies an LTL formula suffices to ensure that the formula holds for ts_1. Therefore, abstraction can drastically shorten verification time.

In this chapter, we lay the theoretical foundations for abstraction-based model checking of SPLs. In Section 5.1, we extend the definition of simulation from TS to FTS and propose an algorithm that computes this relation. We then establish which properties are preserved by the simulation relation, and for which products. This is required to verify abstractions reliably. Then, in Section 5.2 we define three abstractions based on the notion of simulation quotient [16] that can be applied to remove redundant behaviour in an FTS and thus reduce verification time. In addition to abstraction, simulation relations have numerous applications. In particular, simulation-based model checking is an established verification method, as LTL model checking is. Our solution allows easier verification of properties modelled visually (as automata) rather than by logical formulae, which can be more suitable for engineers. Studying FTS simulation is thus as important as generalising LTL model checking to FTS. In Section 5.3, we carry out a complexity evaluation and empirical evaluations that reveal substantial efficiency improvements over enumerative application of TS simulation. This corroborates previous results by characterising the gain of FTS-based simulation over a product-by-product, TS-based, simulation.

5.1 Simulation Relation for SPL Models

Since a model abstraction does not necessarily satisfy the same properties as the original model, it is essential to characterise the behavioural inconsistencies between the model and its abstraction. We previously presented the simulation for TS as a computable relation that can establish this characterisation. However, its definition is clearly unsuitable in our context because we are interested in abstracting SPL models instead of models of individual systems. In this section, we thus introduce a definition of simulation for FTS. We also propose an algorithm to compute it and we establish a link between the relation and the preservation of fLTL formulae.

5.1.1 FTS Simulation Relation

As a first step, we extend the definition of simulation to FTS. For this purpose, we first impose the restriction that a simulation relation can only hold between two FTS defined *over the same FM*. Intuitively, an FTS simulates another one if and only if for every valid product p, the projection of the first FTS onto p is simulated by the projection of the second FTS onto p. For the same reasons we expressed the satisfaction of a formula in terms of products, we want to identify *for which products* an FTS simulates another.

This will allow us to characterise more precisely the preservation of properties in an abstract FTS. Formally, this condition can be expressed using the simulation on TS as defined previously.

Definition 5.2 (Simulation between FTS)
Let fts_1 and fts_2 be two FTS over an FM fm. Then, the products for which fts_2 simulates fts_1 are encoded by the feature expression $(fts_1 \preccurlyeq_{FTS} fts_2)$ such that

$$\forall p \in [\![fm]\!] \bullet (fts_1 \preccurlyeq_{FTS} fts_2)(p) \Leftrightarrow fts_{1\,|p} \preccurlyeq_{TS} fts_{2\,|p}.$$

Furthermore, fts_2 is said to completely simulate fts_1 if and only if $(fts_1 \preccurlyeq_{FTS} fts_2) = fm$.

For recall, since the semantics of an FM is a set of products, we can see it as a feature expression. Moreover, like F-satisfiability, this definition does not consider invalid products.

Thanks to the above definition, we can already determine for which products an FTS simulates another. However, this would require computing the TS simulation relation for $\mathcal{O}(2^{|F|})$ couples of TS, which sums up to an overall time complexity bounded by $\mathcal{O}(|S|^4.2^{|F|})$ [16]. Instead, we aim to take advantage of the compact structure of FTS, as Classen *et al.* did for solving the SPL model-checking problem. Hence, we propose an alternative definition that exploits the variability encoded in the FTS.

Definition 5.3 (F-simulation)
For $i \in \{1, 2\}$, let $fts_i = (S_i, Act_i, Trans_i, I_i, AP, L_i, fm, \gamma_i)$ be two FTS and $Post_i$ be their respective successor function. An F-simulation for (fts_1, fts_2) is a binary function $\mathcal{R} : S_1 \times S_2 \to 2^{(2^F)}$ such that

$$\mathcal{R}(s_1, s_2) = (L_1(s_1) = L_2(s_2)) \wedge \bigwedge_{\alpha, s_1'} \mathcal{R}_{via}(s_1 \xrightarrow{\alpha} s_1', s_2) \tag{5.1}$$

where $\mathcal{R}_{via}(s_1 \xrightarrow{\alpha} s_1', s_2)$ is given by

$$\gamma_1(s_1, \alpha, s_1') \Rightarrow \bigvee_{\beta, s_2'} (\gamma_2(s_2, \beta, s_2') \wedge \mathcal{R}(s_1', s_2'))$$

with $s_j' \in dom(Post_j(s_j))$ for $j \in \{1, 2\}$. Then, fts_2 simulates fts_1 for a valid product p if and only if there exists an F-simulation $\mathcal{R}$ such that

$$p \models \bigwedge_{i_1 \in I_1} \bigvee_{i_2 \in I_2} \mathcal{R}(i_1, i_2).$$

The set of valid products for which fts_2 simulates fts_1 is encoded by the feature expression

$$\bigwedge_{i_1 \in I_1} \bigvee_{i_2 \in I_2} \mathcal{R}_{FTS}(i_1, i_2)$$

where $\mathcal{R}_{FTS}$ is the largest simulation for (fts_1, fts_2). By largest, we mean that for any states s_1, s_2 and simulation $\mathcal{R}$ for (fts_1, fts_2), we have $\mathcal{R}(s_1, s_2) \Rightarrow \mathcal{R}_{FTS}(s_1, s_2)$.

This definition can be seen as a generalisation of Definition 5.1. Intuitively, $[\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$ contains only products whose any behaviour modulo simulation starting from s_1 can be reproduced starting from s_2. In other words, for each product $p \in [\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$ and transition (s_1, α, s_1') available for p, there must exist a transition (s_2, β, s_2') executable by p and such that $p \in [\![\mathcal{R}_{FTS}(s_1', s_2')]\!]$. Similarly, $[\![\mathcal{R}_{via}(s_1 \xrightarrow{\alpha} s_1', s_2)]\!]$ contains only products for which s_2 can simulate the transitions from s_1 to s_1'. Let us note that, according to our definition:

- $\mathcal{R}_{FTS}(s_1, s_2) \not\models \bot$ implies that all atomic propositions satisfied by s_1 are satisfied by s_2 and vice versa.
- Let px and px' be two disjoint sets of products. For given s_1 and s_1' such that $\mathcal{R}_{via}(s_1 \xrightarrow{\alpha} s_1', s_2) = \mathbb{B}(px \cup px')$, it may happen that s_2 simulates a transition $s_1 \xrightarrow{\alpha} s_1'$ for products px via a transition $s_2 \xrightarrow{\beta} s_2'$ and for products px' thanks to another transition $s_2 \xrightarrow{\beta} s_2''$.

We illustrate this second remark through a basic example.

Example 5.1.1 *Let us consider the basic example presented in Figure 5.1. We observe that $\mathcal{R}_{FTS}(1', 2') = f$ and $\mathcal{R}_{FTS}(1', 2'') = \neg f$. Furthermore, state 2 simulates state 1 for products $[\![f]\!]$ through transition $(2, 2')$ and for products $[\![\neg f]\!]$ via transition $(2, 2'')$. In the end, we conclude that state 2 simulates state 1 for all products.*

As established in the following theorem, the two definitions of simulation for FTS we have given are equivalent.

Theorem 5.1 (Equivalence of FTS simulation relations) *Let fts_i, $i \in \{1, 2\}$, be two FTS over the same FM. Let $\mathcal{R}_{FTS}$ be the largest simulation for (fts_1, fts_2). Then, it holds that*

$$(fts_1 \preccurlyeq_{FTS} fts_2) = \bigwedge_{i_1 \in I_1} \bigvee_{i_2 \in I_2} \mathcal{R}_{FTS}(i_1, i_2).$$

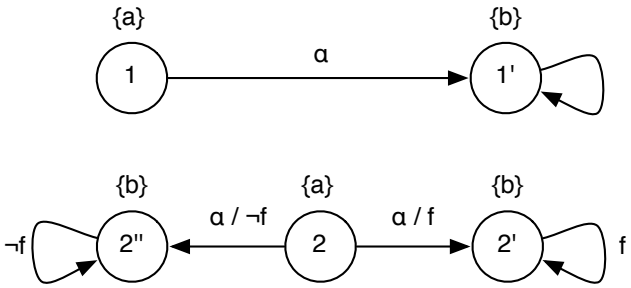


Figure 5.1: Two simulation-equivalent FTS.

PROOF. *By Definition 5.2, we have*

$$\forall p \in (fts_1 \preccurlyeq_{FTS} fts_2) \bullet fts_{1\,|p} \preccurlyeq_{TS} fts_{2\,|p}.$$

According to the above and Definition 5.1, for each product p there exists a simulation relation $\mathcal{R}_{TS}$ such that

$$\forall i_1 \in I_1 \bullet \exists i_2 \in I_2 \bullet \mathcal{R}_{TS}(i_1, i_2)$$

where for all $s_1, s_2 \in S$, $\mathcal{R}_{TS}(s_1, s_2)$ is given by

$$L(s_1) = L(s_2) \wedge \forall s_1' \in \{s' \mid (Post(s_1)(s_1'))(p)\} \bullet \exists s_2' \in \{s' \mid (Post(s_2)(s_2'))(p)\} \bullet \mathcal{R}_{TS}(s_1', s_2'). \quad (5.2)$$

Let us assume that for all p, $\mathcal{R}_{TS}$ is the largest relation satisfying this equation. According to Definition 5.3, we have that

$$p \models \bigwedge_{i_1 \in I_1} \bigvee_{i_2 \in I_2} \mathcal{R}_{FTS}(i_1, i_2)$$

which is equivalent to

$$\forall i_1 \in I_1 \bullet \exists i_2 \in I_2 \bullet p \models \mathcal{R}_{FTS}(i_1, i_2).$$

For a product $p \in [\![\mathcal{R}_{FTS}(s_1, s_2)]\!]$ and any transition (s_1, s_1') available to p, there exists a transition (s_2, s_2') that p can execute such that s_2' simulates s_1' for p, that is $p \models \mathcal{R}_{FTS}(s_1', s_2')$. Hence, $p \models \mathcal{R}_{FTS}(s_1, s_2)$ is equivalent to

$$L(s_1) = L(s_2) \wedge \forall s_1' \in \{s' \mid (Post(s_1)(s_1'))(p)\} \bullet \exists s_2' \in \{s' \mid (Post(s_2)(s_2'))(p)\} \bullet (\mathcal{R}_{FTS}(s_1', s_2'))(p). \quad (5.3)$$

The above results show that the relations $\mathcal{R}_{TS}$ and $\mathcal{R}_{FTS}$ satisfy the same equation (see Equations 5.2 and 5.3). Furthermore, by definition they are

both the largest relation satisfying this equation. By generalizing this result to all products and couples of states, we conclude that

$$(fts_1 \preccurlyeq_{FTS} fts_2) = \bigwedge_{i_1 \in I_1} \bigvee_{i_2 \in I_2} \mathcal{R}_{FTS}(i_1, i_2).$$

□

As for TS, we can define simulation-equivalence for FTS. Intuitively, two FTS are simulation-equivalent for a set of products if and only if they have the same behaviour for all these products.

Definition 5.4 (F-simulation equivalence)
Let fts_1 and fts_2 be two FTSs. Simulation equivalence for fts_1 and fts_2 is the relation

$$(fts_1 \simeq_{FTS} fts_2) = (fts_1 \preccurlyeq_{FTS} fts_2) \wedge (fts_2 \preccurlyeq_{FTS} fts_1).$$

Furthermore, fts_1 and fts_2 are completely simulation-equivalent if and only if $(fts_1 \simeq_{FTS} fts_2) = fm$.

5.1.2 Computing F-Simulation

We proposed two equivalent definitions of the simulation relation for FTS. While the former is more intuitive, it is cumbersome to compute for a large SPL, i.e. with a high number of products. On the contrary, the latter takes advantage of the compact structure of the FTS. In this subsection, we present a method to compute the relation using this second definition. The principles of our algorithm is to compute $\mathcal{R}_{FTS}$ for all couples of states in a given FTS as the smallest fixed point of a function $T : (S \times S \to 2^{(2^F)}) \Rightarrow (S \times S \to 2^{(2^F)})$ that iteratively applies Equation 5.1, that is, $\forall s_1 \in S_1 \bullet s_2 \in S_2$:

$$T(\mathcal{R}(s_1, s_2)) = \bigwedge_{\alpha, s_1'} \big(\gamma_1(s_1, \alpha, s_1') \Rightarrow \mathcal{R}(s_1', s_2') \wedge \bigvee_{\beta, s_2'} \mathcal{R}_{FTS}(s_1', s_2')\big).$$

For the partial order, we consider the implication between Boolean formulae for all couples of states:

$$\forall \mathcal{R} \bullet \forall s_1, s_2 \in S \bullet T(\mathcal{R}(s_1, s_2)) \Rightarrow \mathcal{R}(s_1, s_2).$$

Then according to Knaster–Tarski theorem, $\mathcal{R}_{FTS}$ can be computed as follows:

$$\mathcal{R}_{FTS} = \mathcal{R}_i \bullet \forall j \geq i \bullet \mathcal{R}_i = \mathcal{R}_j$$

where

$$\forall s_1, s_2 \in S \bullet \mathcal{R}_0(s_1, s_2) = \begin{cases} fm, & L(s_1) = L(s_2) \\ \bot, & \text{otherwise} \end{cases}$$
$$\mathcal{R}_{i+1} = T(\mathcal{R}_i).$$

Thanks to the above method we can compute $(fts_1 \preccurlyeq_{FTS} fts_2)$ for any fts_1 and fts_2. For this purpose, we apply it to a composed FTS defined as

$$fts_1 \oplus fts_2 = (S_1 \uplus S_2, Trans_1 \cup Trans_2, Act_1 \cup Act_2, I_1 \cup I_2, AP, L, fm, \gamma)$$

where $\uplus$ denotes the disjoint union, $L(s) = L_i(s)$ for any $s \in S_i$, and $\gamma(t) = \gamma_i(t)$ for any $t \in Trans_i$. Using the value of $\mathcal{R}_{FTS}$ between each initial state of fts_1 and each initial state of fts_2, we determine $(fts_1 \preccurlyeq_{FTS} fts_2)$. Note that Baier and Katoen [16] present a similar method for computing the simulation relation between two TS.

5.1.3 Time Complexity

The time complexity of the above computation method is evaluated in the following theorem.

Theorem 5.2
The time complexity of computing the simulation function is bounded by $\mathcal{O}(|S|^6.2^{3.|F|})$, *where* F *is the set of features.*

PROOF. *Let* k *be the smallest such that* $\forall j > k \bullet \mathcal{R}_k = \mathcal{R}_j$. *For* $i < k$, *there is at least one triplet* $(s_1, s_2, p) \in S \times S \times [\![fm]\!]$ *such that*

$$p \models \mathcal{R}_i(s_1, s_2) \wedge \neg\mathcal{R}_{i+1}(s_1, s_2).$$

Consequently, $k \leq |S|^2.2^{|F|}$.

Assume we represent each feature expression $\mathcal{R}(s_1, s_2)$ *by a Binary Decision Diagram (BDD) [32]. It is at most of size* $\mathcal{O}(2^{|F|})$. *Computing* $\mathcal{R}_{i+1}$ *is a conjunction or disjunction on pairs of transitions. These operations are quadratic in the size of the BDD. Since the number of conjunction and disjunction is bounded by* $\mathcal{O}(|Trans|^2)$, *each step takes* $\mathcal{O}(|Trans|^2.2^{2.|F|}) \leq \mathcal{O}(|S|^4.2^{2.|F|})$. *To verify if the fixed point has been reached, we must determine if, for all* $(s_1, s_2) \in S \times S$, $\mathcal{R}_i(s_1, s_2) \leq \mathcal{R}_{i+1}(s_1, s_2)$. *Establishing this*

comes to checking if $\mathcal{R}_i(s_1, s_2) \wedge \neg\mathcal{R}_{i+1}(s_1, s_2)$ *is unsatisfiable, which is of cost* $|S|^2.2^{2.|F|}$.

The total time complexity of computing $\mathcal{R}_{FTS}$ *is thus bounded by*

$$\mathcal{O}(|S|^6.2^{3.|F|}).$$

□

Although it is theoretically dominated by $2^{3.|F|}$, and thus in EXPTIME, in practice $|S|^6$ is often bigger.

5.1.4 Property Preservation

The simulation relation for TS is particularly well-known for its interesting preservation properties [144, 45, 16]. In particular, if TS_2 simulates TS_1, then any LTL formula satisfied by TS_2 is also satisfied by TS_1. As we show in this section, a similar results holds for F-simulation and fLTL.

Theorem 5.3
Let fts_1 *and* fts_2 *be two FTS over the same FM,* $\psi = [\chi]\phi$ *an fLTL formula, and* $px = [\![fts_1 \preccurlyeq_{FTS} fts_2]\!]$. *Then, it holds that*

$$(fts_1 \preccurlyeq_{FTS} fts_2) \Rightarrow (fts_2 \models \psi \Rightarrow fts_1 \models \psi). \tag{5.4}$$

Proof. *Let us first remark that*

$$\neg\chi(p) \Rightarrow ((fts_1 \models \psi)(p) \Leftrightarrow (fts_2 \models \psi)(p)).$$

The right-hand side of Equation 5.4 is thus trivially satisfied for products not satisfying χ. *Let us now consider a product* p *satisfying* χ. *By Definition 5.2, we have*

$$(fts_1 \preccurlyeq_{FTS} fts_2)(p) \Leftrightarrow fts_{2\,|p} \preccurlyeq_{TS} fts_{1\,|p}.$$

Furthermore, by Property 5.1, we know that

$$fts_{2\,|p} \models \phi \Rightarrow fts_{1\,|p} \models \phi.$$

and by definition of F-satisfiability (see Definition 3.6), we have

$$(fts_i \models \psi)(p) \Leftrightarrow fts_{i\,|p} \models \phi.$$

Thus, it holds that $(fts_2 \models \psi)(p) \Rightarrow (fts_1 \models \psi)(p)$. □

The above theorem also implies that two completely simulation-equivalent FTS have the same F-(un)satisfiability for any fLTL formula.

5.2 FTS Simulation Quotient

In the previous section, we defined a simulation relation for FTS and we established the link between this relation and the F-satisfiability of an fLTL formula. Our next objective is to study simulation quotient, which leads to the computation of preliminary FTS abstractions. We also make use of the preservation properties (see Theorem 5.3) to determine how the verification of an abstract FTS provides information about the original model.

5.2.1 Simulation Quotient

The first abstraction we introduce does not modify the behaviour of the FTS to which it is applied. It merely consists in defining the simulation quotient [16] for FTS. This form of abstraction merges states that are completely simulation-equivalent, i.e. for all valid products. Formally, we define the binary relation

$$\simeq^{fm}_{FTS} \subseteq S \times S \bullet s_1 \simeq^{fm}_{FTS} s_2 \Leftrightarrow \big((\mathcal{R}_{FTS}(s_1, s_2) \wedge \mathcal{R}_{FTS}(s_2, s_1)) = fm\big).$$

This relation is an equivalence, since $\simeq_{TS}$ is also an equivalence relation [16]. Therefore, the state space of any FTS can be partitioned into equivalence classes under $\simeq^{fm}_{FTS}$. Our first abstraction function merges states of the same equivalence class.

Let $[s]_{/\simeq^{fm}_{FTS}}$ denote the equivalence class of s under $\simeq^{fm}_{FTS}$. The abstraction associates an FTS fts with an abstract FTS $fts_{/\simeq^{fm}_{FTS}} = (S', \{\tau\}, Trans', I', AP, L', d, \gamma')$ such that

$$\begin{aligned}
S' &= \{[s]_{/\simeq^{fm}_{FTS}}\} \\
Trans' &= \{([s]_{/\simeq^{fm}_{FTS}}, \tau, [s']_{/\simeq^{fm}_{FTS}}) \\
&\qquad \mid s' \in dom(Post(s))\} \\
I' &= \{[i]_{/\simeq^{fm}_{FTS}} \mid i \in I\} \\
L'([s]_{/\simeq^{fm}_{FTS}}) &= L(s) \\
\gamma'(([s]_{/\simeq^{fm}_{FTS}}, \tau, [s']_{/\simeq^{fm}_{FTS}})) &= \bigvee_{(s'',\alpha,s''') \in Trans} \gamma(s'', \alpha, s''')
\end{aligned}$$

with $s'' \in [s]_{/\simeq^{fm}_{FTS}}$, $s''' \in [s']_{/\simeq^{fm}_{FTS}}$. It thus requires to compute first the simulation relation for every pair of states in the FTS (see Section 5.1.2).

Such an abstract FTS has exactly the same behaviour as the FTS on which the abstraction is applied. Indeed, we merge only states that are simulation-equivalent for every valid product. Thus, the merging neither adds nor removes any behaviour.

Theorem 5.4
Let fts be an FTS over an FM fm. Then, we have

$$(fts \simeq_{FTS} fts_{/\simeq^{fm}_{FTS}}) = fm.$$

PROOF. *The proof consists in showing that for any s_1 we have*

$$\mathcal{R}_{FTS}(s_1, [s_1]_{/\simeq^{fm}_{FTS}}) = fm \tag{5.5}$$
$$\mathcal{R}_{FTS}([s_1]_{/\simeq^{fm}_{FTS}}, s_1) = fm. \tag{5.6}$$

Obviously, $\mathcal{R}_{FTS}(s_1, [s_1]_{/\simeq^{fm}_{FTS}}) = fm$ holds since any transition from s_1 to a state s'_1 in fts is simulated for all products by a transition from $([s_1]_{/\simeq^{fm}_{FTS}}$ to $[s'_1]_{/\simeq^{fm}_{FTS}})$, by definition of $fts_{/\simeq^{fm}_{FTS}}$.

Let us now prove that $\mathcal{R}_{FTS}([s_1]_{/\simeq^{fm}_{FTS}}, s_1) = fm$ also holds by contradiction. Assume it does not hold, that is,

$$\exists [s_1]_{/\simeq^{fm}_{FTS}} \xrightarrow{\tau} [s'_2]_{/\simeq^{fm}_{FTS}} \bullet \big(\gamma([s_1]_{/\simeq^{fm}_{FTS}}, \tau, [s'_2]_{/\simeq^{fm}_{FTS}}) \Rightarrow \bigvee_{\alpha_1} \gamma(s_1, \alpha_1, s'_1) \wedge \mathcal{R}_{FTS}([s'_2]_{/\simeq^{fm}_{FTS}}, s'_1)\big) \neq fm.$$

This implies that

$$\exists s_2 \in [s_1], s'_2 \bullet s_2 \xrightarrow{\alpha_2} s'_2 \wedge \big(\gamma(s_2, \alpha_2, s'_2) \Rightarrow \bigvee_{\alpha_1} \gamma(s_1, \alpha_1, s'_1) \wedge \mathcal{R}_{FTS}(s'_2, s'_1)\big) \neq fm.$$

and thus that $\mathcal{R}_{FTS}(s_2, s_1) \neq fm$. However, this is impossible since

$$\{s'_1, s_2\} \subseteq [s_1] \Leftrightarrow \mathcal{R}_{FTS}(s_1, s_2) = fm \wedge \mathcal{R}_{FTS}(s_2, s_1) = fm.$$

□

In spite of its straightforward computation, this first abstraction function has shown to be inefficient when it comes to actually reducing the state-space, as we will see in Section 5.3. In what follows, we define coarser abstraction methods based on F-simulation quotient that still preserve properties.

5.2.2 Reachability-Aware Simulation Quotient

The second abstraction method is similar to the first one, but it takes into account the reachability of each state when determining the equivalence classes. It requires the computation of a function $\preccurlyeq_R \bullet S \times S \rightarrow 2^{(2^F)}$. In simple terms, $s_1 \preccurlyeq_R s_2$ gives a feature expression satisfied by the products for which s_1 and s_2 simulate each other, while considering the reachability function associated with s_1 and s_2 respectively. More precisely, we define the following:

- s_2 trivially simulates s_1 for products that cannot reach s_1;
- s_2 cannot simulate s_1 for products that can reach s_1 but not s_2.

Accordingly, $(s_1 \preccurlyeq_R s_2)$ is defined as

$$R(s_1) \Rightarrow (R(s_2) \wedge \mathcal{R}_{FTS}(s_1, s_2))$$

where $R(s)$ denotes the feature expression satisfied by the products that can reach state s. Next, we define the binary relation $\simeq_R \subseteq S \times S$ such that $s_1 \simeq_R s_2$ if and only if $(s_1 \preccurlyeq_R s_2) \wedge (s_2 \preccurlyeq_R s_1) = fm$. Again, $\simeq_R$ is an equivalence relation. It is obviously reflexive and symmetric. Its transitivity is demonstrated in the following theorem.

Theorem 5.5
Let fts be an FTS and s_1, s_2, s_3 be three of its states. Then it holds that

$$s_1 \simeq_R s_2 \wedge s_2 \simeq_R s_3 \Rightarrow s_1 \simeq_R s_3$$

PROOF. *Let us note that $s_1 \simeq_R s_2 \wedge s_2 \simeq_R s_3$ is equivalent to*

$$(\neg R(s_1) \vee (R(s_2) \wedge \mathcal{R}_{FTS}(s_1, s2)) \ \wedge (\neg R(s_2) \vee (R(s_3) \wedge \mathcal{R}_{FTS}(s_2, s_3))$$

which can be re-written as

$$(\neg R(s_1) \wedge (\neg R(s_2) \vee R(s_3) \wedge \mathcal{R}_{FTS}(s_2, s_3)) \\ \vee (R(s_2) \wedge \mathcal{R}_{FTS}(s_1, s_2) \wedge R(s_3) \wedge \mathcal{R}_{FTS}(s_2, s_3))$$

Since the following hold:

$$\neg R(s_1) \wedge (\neg R(s_2) \vee R(s_3) \wedge \mathcal{R}_{FTS}(s_2, s_3)) \Rightarrow \neg R(s_1)$$
$$R(s_2) \wedge R(s_3) \Rightarrow R(s_3)$$
$$\mathcal{R}_{FTS}(s_1, s_2) \wedge \mathcal{R}_{FTS}(s_2, s_3) \Rightarrow \mathcal{R}_{FTS}(s_1, s_3)$$

the above expression implies $s_1 \simeq_R s_3$. □

Consequently, the state space of an FTS can be partitioned into equivalent classes under $\simeq_R$. Using this binary relation, we define an abstraction function that merges the states of an FTS according to their equivalence class under $\simeq_R$. Hence, the results of applying the function on an FTS fts is an abstract FTS $fts_{/\simeq_R}$, which is defined similarly to $fts_{/\simeq^{fm}_{FTS}}$.

We can show that this abstract FTS has exactly the same behaviours as the original one modulo F-simulation, that is, $fts \simeq_{FTS} fts_{/\simeq_R}$. The idea of the proof is the following. Let $p \in [\![fm]\!]$ be a product. If $s_1 \simeq_R s_2$, then p can reach either both s_1 and s_2 or none of them:

1. If p can reach both s_1 and s_2 then its behaviour starting from s_1 is equivalent to that starting from s_2, by definition of $\simeq_R$ and $\mathcal{R}_{FTS}$. Therefore, merging s_1 and s_2 would not add any behaviour to p.
2. If p can reach neither s_1 nor s_2, then merging the two would not actually add behaviour to p since it would not be able to reach the resulting abstract state.

This result is formally established in the following theorem.

Theorem 5.6
Let fts be an FTS over an FM fm. Then $(fts \simeq_{FTS} fts_{/\simeq_R}) = fm$.

PROOF. *By definition of $fts_{/\simeq_R}$, we trivially have*

$$(fts \preccurlyeq_{FTS} fts_{/\simeq_R}) = fm.$$

Now, assume that $(fts_{/\simeq_R} \preccurlyeq_{FTS} fts) = fm$. For this purpose, we prove that a state merging does not add behaviour to fts modulo F-simulation.

Let s_1 and s_2 such that $s_1 \simeq_R s_2$. The two states are thus in the same equivalence class. Without loss of generality, we assume that $[s_1]_{\simeq_R} = \{s_1, s_2\}$. By definition of $\simeq_R$, it holds that $R(s_1) = R(s_2)$. Let $p \in [\![fm]\!]$. Two cases can occur:

1. *$R(s_1)(p)$ does not hold. Then any trace $s'_0, s'_1, \cdots \in [\![fts_{|p}]\!]$ is such that $s'_i \neq s_1$ and $s'_i \neq s_2$ for any i. Therefore, $s'_i \neq [s_1]_{\simeq_R}$ also holds by definition of $fts_{/\simeq_R}$. Hence, the merging of s_1 and s_2 has no impact on the behaviour of p.*
2. *$R(s_1)(p)$ holds. In this case, $\mathcal{R}_{FTS}(s_1, s_2)(p) \wedge \mathcal{R}_{FTS}(s_2, s_1)$. Similarly to Theorem 5.4, we can prove that $\mathcal{R}_{FTS}([s_1]_{\simeq_R}, s_1)(p)$ holds.*

Hence, $fts_{/\simeq_R} \preccurlyeq_{FTS} fts) = fm$ and this concludes the proof. □

5.2.3 Reachability-Aware Preorder-Based Abstraction

Unlike the previous ones, the last abstraction we introduce in this chapter illustrates that abstraction can lead to alterations in the FTS behaviour. In this case, new behaviour is added. Informally, for any couple of states (s_1, s_2), if $(s_1 \preccurlyeq_R s_2) = fm$, then s_1 is integrated into s_2. By integration, we mean that all the transitions going to s_1 are redirected to s_2, and s_2 as well as its outgoing transitions are discarded. Since s_2 simulates s_1 for any product, we do not remove any behaviour from the FTS. However, new behavioural options may appear for products in $[\![fm]\!] \cap [\![\neg(s_2 \preccurlyeq_R s_1)]\!]$.

Theorem 5.7
Let s, s' be states of an FTS fts such that $(s \preccurlyeq_R s') = fm$, $\neg(s' \preccurlyeq_R s) \neq \bot$. Then integrating s into s' results in an FTS fts' such that $(s' \preccurlyeq_R s) \Rightarrow (fts' \preccurlyeq_{FTS} fts)$ is not a tautology.

PROOF. *Let $s' \in S \bullet \nexists s \in S \bullet s' \preccurlyeq_R s$, and $s \in S \bullet s \preccurlyeq_R s'$. Let $p \in [\![fm]\!] \bullet R(s)(p)$. Then $R(s')(p)$ holds by definition of $\preccurlyeq_R$. Let π be a trace starting from s' that s cannot produce.*

Now assume that there is a trace prefix $A_0, \ldots, A_n, L(s)$ starting from an initial state s_0 and leading to s such that there exist no equivalent trace prefix leading to s'. The trace $(A_0, \ldots, A_n)\pi$ exists in fts' but not in fts. Therefore, $(fts' \preccurlyeq_{FTS} fts)(p)$ does not hold. □

Let us observe that this form of abstraction is not a function, since for a given FTS it may lead to more than one abstract FTS. For example, let s_1, s_2, s_3 be three states such that $(s_1 \preccurlyeq_R s_2) = (s_1 \preccurlyeq_R s_3) = fm$, $(s_2 \preccurlyeq_R s_3) \neq fm, (s_3 \preccurlyeq_R s_2) \neq fm$ and there exists no s_4 such that $(s_2 \preccurlyeq_R s_4) = (s_3 \preccurlyeq_R s_4) = fm$. This implies that s_1 can be integrated into either s_2 and s_3, but these two will never be integrated one into the other, or into another state.

Instead of defining formally the set of FTS that can result from one of these abstractions, we give an algorithm to greedily build one of its element (see Algorithm 2). First, we register the couples of states (s_1, s_2) such that $(s_1 \preccurlyeq_R s_2) = fm$ in a set R (Line 1). Next, we keep merging states as much as possible (Lines 2-14). At each iteration, we remove an element of R (and S) non-deterministically (Lines 3-5). Let (s_1, s_2) be this element. Then, s_1 is not part of the state space of the abstract FTS (Line 5). Furthermore, if s_1 was an initial state, then s_2 becomes an initial state of the abstract FTS (Lines 6-8). As mentioned earlier, each transition of the form $(s, s_1), s \neq s_1$, is transformed into a transition (s, s_2) and the new transition-labelling function γ' is modified accordingly (Lines 9-13).

Input: An FTS $(S, Act, Trans, I, AP, L, fm, \gamma)$
Output: An abstract FTS $\widehat{fts}$ smaller than fts and such that $(fts \preccurlyeq_{FTS} \widehat{fts}) = fm$.

$R \leftarrow \{(s_1, s_2) \mid s_1 \neq s_2 \wedge (s_1 \preccurlyeq_R s_2) = fm\}$;
while $R \neq \emptyset$ **do**
  Let $(s_1, s_2) \in R$;
  $R \leftarrow R \setminus \{(s, s') \in R \mid s = s_1 \vee s' = s_1\}$;
  $S = S \setminus \{s_1\}$;
  if $s_1 \in I$ **then**
    $I \leftarrow (I \cup \{s_2\}) \setminus \{s_1\}$;
  end
  $Remove \leftarrow \{(s, s') \in Trans \mid s = s_1 \vee s' = s_1\}$;
  $Trans \leftarrow (Trans \setminus Remove) \cup \{(s, s_2) \mid s \neq s_1 \wedge (s, s_1) \in Remove\}$;
  foreach $s : \{(s, s_1), (s, s_2)\} \subseteq Trans$ **do**
    $\gamma'(s, s_2) \leftarrow \gamma(s, s_1) \vee \gamma(s, s_2)$;
  end
end
return $(S, \{\tau\}, Trans, I, AP, L, fm, \gamma')$

Algorithm 2: Computation of the simulation function

5.3 Preliminary Experiment

This section describes preliminary experiments we conducted to evaluate the time and space gain obtained thanks to the above abstraction methods.

5.3.1 Experiment Setup

To carry out these experiments, we have integrated the computation of F-simulation as well as the three abstractions into an Haskell FTS library[1] developed by Classen *et al.* [59]. It allows us to validate our approach and to measure its efficiency when it comes to computing the simulation relation and reducing both the state-space size of an FTS and its verification time. All benchmarks were run on a MacBook Pro with a 2,4 GHz Core 2 Duo processor and 4 Gb of RAM. The library was compiled using the Glasgow Haskell Compiler.[2] To avoid the influence of other running processes, we repeated each experiment 10 times.

Our evaluation considers the mine pump controller SPL defined in [59]. The whole system is designed as the parallel composition of several processes (a pump, a water sensor, a methane sensor, and a controller). The mine pump SPL has nine features and 64 products. The FTS modelling

[1] https://projects.info.unamur.be/fts/implementations/haskell-library/
[2] http://www.haskell.org/ghc/

	Def. 5.2	Def. 5.3
$m_{base} \preccurlyeq_{FTS} m_{ext}$	3247.07	113.39
$m_{ext} \preccurlyeq_{FTS} m_{base}$	3150.97	108.59
Total	6398.04	221.98

Table 5.1: Computation time of F-simulation (in seconds)

its behaviour, noted m_{base} is composed of 465 states and 1306 transitions (see [55] for a detailed description).

5.3.2 Evaluation of F-Simulation Computation

We introduced two methods for computing F-simulation for two FTS. The former is based on a product-by-product approach and determine, for given fts_1 and fts_2, and each product p, if $fts_{1\,|p} \preccurlyeq_{TS} fts_{2\,|p}$. The latter makes use of the compact structure of FTS and is based on the computation of a fixed point, as stated in Subsection 5.1.2. Our first experiments evaluate the practical efficiency of both methods.

The evaluation considers the minepump system, m_{base}, as well as an extension of it. Basically, we extended the behavioural options of some of the products by making them able to execute additional transitions. This results in an extended model, noted m_{ext}. Then, we measured the time needed by both methods to compute $m_{base} \preccurlyeq_{FTS} m_{ext}$ and $m_{ext} \preccurlyeq_{FTS} m_{base}$. Benchmarks results are shown in Table 5.1.

We observe that the algorithm based on Definition 5.3 is far more efficient than the one that enumerates the products and computes the TS-simulation of their projection. In spite of having a worse theoretical time complexity, it is 28.82 times faster than the product-based algorithm. Note that the execution time of the latter includes the time needed for determining the projection of each product, which amounts to about 20% of the whole execution time.

5.3.3 Evaluation of Temporal Property Verification

Our second evaluation benchmarks the time needed for model checking abstractions combined with either the product-by-product approach or the FTS algorithms. The objective of the following experiments is to determine which abstraction method is more efficient. For the former method, we evaluate the verification time by enumerating all the products and computing their projection on (1) the original model, (2) its TS-simulation quotient $m_{\simeq_{TS}}$ [16], and (3) the model $m_{\preccurlyeq_{TS}}$, obtained by greedily integrating a state s_1 into another s_2 if and only if $s_1 \preccurlyeq_{TS} s_2$.

Also, we apply FTS abstractions to obtain three abstract FTS $m_{\simeq_{FTS}}$,

Enumerative method

Formula	m_{base}		$m_{\simeq_{TS}}$		$m_{\preccurlyeq_{TS}}$	
#1	31.23	✗	40.42	✗	39.70	✗
#2	9.79	✓	19.58	✓	19.75	✓
#3	134.71	✗	178.52	✗	175.11	✗
#4	8.38	✓	17.2	✓	17.97	✓
#5	19.18	✓	32.22	✓	33.51	✗
#6	27.57	✗	37.94	✗	36.69	✗
#7	9.44	✓	19.37	✓	20.26	✓
#8	46.19	✗	64.48	✗	62.22	✗
#9	11.39	✗	24.58	✗	25.64	✗
#10	8.78	✓	19.54	✓	19.25	✓
Total	**306.66**		**453.85**		**450.1**	

Table 5.2: Verification time using the enumerative method (in seconds)

$m_{\simeq_R}$, and $m_{\preccurlyeq_R}$ respectively. For the latter, when a state can be integrated into more than one state, our choice is based on the lexicographic order. For example, if we have $s_1 \preccurlyeq_{FTS} s_2 = s_1 \preccurlyeq_{FTS} s_3 = fm$ then we integrate s_1 into s_2 rather than in s_3.

For each FTS, we first compute the number of states and transitions in order to determine to what extent a given abstraction reduces the size of the original FTS. We observe that the abstraction based on the equivalence classes under $\simeq_{FTS}$ yields no reduction at all. Its merging condition seems too restrictive in the context of product lines. Since an FTS models the behaviour of $\mathcal{O}(2^{|F|})$ products, it is very unlikely that two states have exactly the same behavioural options for all those products.

Taking into account the reachability already allows to merge states, although only a few of them. The state-space size is thus reduced to 459 states and 1284 transitions. Finally, the third abstraction yields a reduction of about 9% (423 states and 1192 transitions). Although these are the best results in terms of state-space reduction, we must keep in mind that, like every efficient state-space reduction method, it augments the behaviour of the FTS. Hence, we may find false negatives, i.e. products that violate a given property in the abstract FTS but not in the original one.

In order to evaluate the impact of the state-space reduction on the verification time, we model-checked the seven models against ten different properties expressed in LTL, such as those defined in [6] and [59]. The results are shown in Tables 5.2 and 5.3. For every formula and every model, we give the time needed to verify the model against the formula. We also describe if the formula is verified by every valid product (✓) or not (✗). The verification

Formula	m_{base}		$m_{\simeq_R}$		$m_{\preccurlyeq_R}$	
#1	13.08	✗	13.46	✗	12.07	✗
#2	0.81	✓	2.15	✓	1.94	✓
#3	97.91	✗	92.37	✗	82.23	✗
#4	0.96	✓	2.31	✓	2.01	✓
#5	1.26	✓	2.56	✓	2.29	✗
#6	10.10	✗	10.89	✗	9.91	✗
#7	0.41	✓	1.79	✓	1.62	✓
#8	7.2	✗	7.7	✗	6.80	✗
#9	0.65	✗	2.02	✗	1.79	✗
#10	0.49	✓	1.86	✓	1.67	✓
Total	132.87		137.11		122.33	

Table 5.3: Verification time using the FTS algorithms (in seconds)

time of every property includes the computation time of the abstractions, which represents about 10% of the overall verification time, in both cases. This overhead could be partially avoided if the abstractions are computed once and for all. We do not present the verification times for $m_{\simeq_{FTS}}$. Since it has as many states and transitions as m_{base}, any difference would be the result of random variations independent of the verification process.

Let us first discuss the results for the product-based and the FTS approaches separately. When summing up all the times related to the former, we observe that the overhead due to the computation of both the abstractions based on TS-simulation quotient ($m_{\simeq_{TS}}$) and preorder ($m_{\preccurlyeq_{TS}}$) is significant. Because of that, the verification times of $m_{\simeq_{TS}}$ and $m_{\preccurlyeq_{TS}}$ are respectively 48% and 47% higher than the model-checking time of m_{base} using the enumerative method. Furthermore, false negatives appeared when checking $m_{\preccurlyeq_{TS}}$ against formula #5. This formula is supposed to be satisfied by all products, but one of them violates it in $m_{\preccurlyeq_{TS}}$ due to the addition of behaviour.

Applying abstraction to FTS yields better results. The abstraction under $\simeq_R$ increases the checking time of m_{base} with FTS by 3%. This verification time decreases by 8% when $m_{\preccurlyeq_R}$ is model-checked. However, it has to be noted that false negatives were found for formula #5. If we compare these results with the ones of the product-by-product methods, we conclude that the FTS-based approaches outperform the enumerative ones. Furthermore, applying abstraction to an FTS can reduce its verification cost. On the contrary, combining a product-based method with an abstraction function is inefficient. This corroborates the claims of Classen *et al.* that such a method should be avoided [53, 56].

Although abstraction clearly permits to reduce the verification time of

an FTS, we are aware that the gain is not significant. This illustrates the difficulty to find a good abstraction for FTS, a formalism that models the behaviours of a potentially large number of systems. A good abstraction should either add behaviour or remove some, but not both. Otherwise, we would not be able to infer any property of the system using the verification results of its abstract counterpart, since a property may be violated by an additional behaviour or satisfied thanks to the removal of an existing one. Therefore, it is particularly difficult to find a state merging condition that both satisfies this requirement and makes a significant reduction. In particular, the purpose of simulation quotient is to eliminate redundancy, not to produce coarse abstractions. More research is required to find ways to design efficient abstraction functions and to finely evaluate their merits with respect to verification performance and false negatives they induce. The current abstractions are applied directly on the FTS itself. The most successful applications of abstraction, like partial-order reduction [96], statement merging [114], and predicate abstraction [48] make use of additional information like parallelism, scope of variables, and conditional statements. These information are not found in such a fundamental formalism, but instead in high-level languages. We study such advanced abstractions in Chapter 6.

Nevertheless, it is interesting to observe that the most significant speed-ups occur during the verification of the most time-consuming properties. This indicates that abstraction can improve the scalability of SPL verification. Naturally, this early indication needs to be confirmed by further experiments. These results combined with previous experiments [59, 58, 55] confirm that FTS is a viable approach for verifying variability-intensive systems.

5.3.4 Threats to Validity

Several threats to the validity of our conclusions have to be pointed out. First, our evaluation is solely based on one case. Other systems of different size and variability should be considered in order to analyse how the different approaches scale with the number of features and the size of the state-space.

This case study has been implemented in Haskell, a programming language that makes use of the so-called lazy evaluation. It means that a value is computed only when it is needed. Although this evaluation method may have influenced our results, we believe that the conclusions would remain valid if we used another programming language.

The comparison between the product-based methods and the FTS algorithms is based on the verification of all the products. However, when a property (or a simulation relation) is required to hold for the whole SPL, we could stop the checking process as soon as a bad product is found. Even so, this comes to the standard model checking problem, and we are interested in identifying all the products that violate a property.

Also, we obtain the verification times related to the product-by-product methods by summing up the verification times for each product individually. In practice, it is very unlikely that those products are verified sequentially, without taking advantage of multi-threading and parallel verification.

5.4 Simulation-Based Feature Characterisation

An inherent characteristic of Classen *et al.*'s algorithms is that they consider all features altogether. As opposed to theirs, there exist methods for *compositional reasoning* on features (see more in Section 4.3). Their idea is to consider features as modifications to an existing system, henceforth named the *base system*. From a model-checking perspective, it means that features are defined as additions and removals of states or transitions. This way of modelling allows a compositional verification of features, where the effect of features on property satisfaction is determined according to the verified base system and the compositional definition of the feature.

On the contrary, Classen *et al.*'s approach does not consider features in isolation. It implies that a feature cannot be verified separately from the others and that the complexity of the FTS algorithms remains exponential in the number of features. Moreover, if a new feature appears in a previously model-checked SPL, the results of earlier verifications cannot be reused. In this section, we address this last issue and lay the foundations towards a progressive and modular verification of SPLs. We identify two special classes of features, the *conservative* [36] and the *regulative* [105] features. Intuitively, a conservative (resp. regulative) feature does not remove (resp. add) any behaviour to the products of an SPL. We first define those features formally and from a model-checking perspective. Secondly, we introduce a formal method based on F-simulation to determine if a feature is conservative or regulative in a given SPL. Finally, we demonstrate that when such particular features appear in an SPL, we can avoid verifying a subset of the products it adds to the SPL.

5.4.1 Conservative and Regulative Features

As a first step, we define two types of features called *conservative* [36] and *regulative* [105]. Intuitively, a feature is conservative for a product p if any behaviour of p still exists once the feature is added to p. On the contrary, the addition of a regulative feature to p does not create new behaviour. We can formally define such features using the projection operator and the simulation relation for TS.

Definition 5.5 (Conservative Feature)
Let fts be an FTS over an FM fm and f be a feature of fm. Then, f is conservative for a product $p \in [\![fm]\!]$ if and only if

$$p \cup \{f\} \models fm \Rightarrow fts_{|p} \preccurlyeq_{TS} fts_{|p \cup \{f\}}.$$

Moreover, f is completely conservative if and only if it is conservative for all valid products.

Definition 5.6 (Regulative Feature)
Let fts be an FTS over an FM fm and f be a feature of fm. Then, f is regulative for a product $p \in [\![fm]\!]$ if and only if

$$p \cup \{f\} \in [\![fm]\!] \Rightarrow fts_{|p \cup \{f\}} \preccurlyeq_{TS} fts_{|p}$$

Moreover, f is completely regulative if and only if it is regulative for all valid products.

Note that a feature is conservative and regulative by default for a given product if it cannot be added to this product, that is, if the resulting new product is not valid according to the FM. Also, the addition of feature that is both conservative and regulative for a product p does not modify the behaviour of p. In this case, the feature is called *spectative* for p. Accordingly, features that cannot be added to p are spectative for it.

Determining if a feature is conservative or regulative is of great interest. For instance, consider that a new feature f is implemented within an already verified SPL. To answer the SPL model-checking problem, one has to verify any valid product introduced by f. Yet if the feature is conservative then it is not needed to verify all these products. Before elaborating on this point, we first focus on the definition of these features.

5.4.2 Restriction Operator

As for FTS model checking and F-simulation, we want to avoid using a product-by-product method to determine if a feature f is conservative or regulative. Any implementation based on Definitions 5.5 and 5.6 would thus not be suitable. Instead, we propose a method based on F-simulation. Basically, we generalize the projection operator in order to obtain an FTS modelling only a subset of the products. In particular, we apply this new

operator to extract an FTS modelling only the products having f. Then, using F-simulation we compare these products with respect to those that do not include f. We name this operator *restriction*.

Definition 5.7 (Restriction of FTS)
Let $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ be an FTS over a set of feature F and $\rho : F \to \{\top, \bot\}$ be a feature valuation function, i.e. a (possibly partial) function that associates features with a Boolean value. Then, fts restricted to ρ, noted $fts^{[\rho]}$, is an FTS $(S, Act, Trans', I, AP, L, d, \gamma')$ such that

$$\gamma(t)^{[\rho]} \models \bot \Rightarrow t \notin Trans'$$
$$\gamma(t)^{[\rho]} \not\models \bot \Rightarrow (t \in Trans' \wedge \gamma'(t) = \gamma(t)^{[\rho]})$$

where $\gamma(t)^{[\rho]}$ is $\gamma(t)$ with all occurrences of any $f \in dom(\rho)$ replaced by $\rho(f)$. Furthermore, a product p is compatible with ρ if and only if $f \in dom(\rho) \Rightarrow (f \in p \Leftrightarrow \rho(f))$.

In simple terms, $fts^{[\rho]}$ is obtained from fts after replacing, in all feature expressions, a feature $f \in dom(\rho)$ by $\top$ (respectively, $\bot$) if and only if $\rho(f)$ holds (respectively, does not hold). In particular, if $\rho(f)$ holds then any product p behaves in the new FTS as the product $p \cup \{f\}$ in the original FTS. We can see the projection as a particular case of restriction where ρ is total. This result, formally established in the following theorem, is an essential step towards proving our approach.

Theorem 5.8
Let fts be an FTS over an FM fm with a set of features F, $\rho : F \to \{\top, \bot\}$, and $p \in [\![fm]\!]$ compatible with ρ. Then $fts_{|p} = fts^{[\rho]}{}_{|p}$ where $=$ denotes syntactic equivalence.

PROOF. *According to Definitions 3.3 and 5.7, both projection and restriction operators modify only the set of transitions and their associated feature expressions.*

Let $Trans$, $Trans'$, and $Trans''$ be respectively the set of transitions in fts, $fts_{|p}$, and $fts^{[\rho]}{}_{|p}$. The proof consists in showing that $Trans'$ and $Trans''$ are equal. According to Definitions 3.3 and 5.7, we have that

$$Trans' = \{t \in Trans \mid \gamma(t)(p)\}$$
$$Trans'' = \{t \in Trans \mid \gamma(t)^{[\rho]}(p)\}$$

More generally, we show that for any $\gamma \in 2^{(2^F)}$, product p and feature valuation ρ, we have $\gamma(t)(p) \Leftrightarrow \gamma(t)^{[\rho]}(p)$. Let us first remark that p can be regarded as a total feature valuation function. Henceforth we interchangeably use the notation $p(f)$ (respectively $\neg p(f)$) in place of $f \in p$ (respectively $f \notin p$). It holds that $\gamma(p) \Leftrightarrow \gamma^{[p]} \not\models \bot$. For the same reason, we have $\gamma(p)^{[\rho]} \Leftrightarrow (\gamma^{[\rho]})^{[p]} \not\models \bot$. Furthermore, since by hypothesis $p(f) = \rho(f)$ for any $f \in dom(\rho)$ it also holds that $\gamma^{[p]} \Leftrightarrow (\gamma^{[\rho]})^{[p]}$. By transitivity of bi-implication, we obtain $\gamma(t)(p) \Leftrightarrow \gamma(t)^{[\rho]}(p)$. □

The proof of this theorem also shows that the projection operator can be derived from the restriction operator. Indeed, for any feature expression γ, $\gamma(p)$ is equivalent to $\gamma^{[p]} \not\models \bot$. Thus, $fts_{|p}$ and $fts^{[p]}$ have the same transition set. Moreover, by Definition 5.7 the feature expression of any transition of $fts^{[p]}$ is a tautology. Hence the projection of $fts^{[p]}$ onto any valid product is syntactically equivalent to $fts_{|p}$.

Another property of the restriction operator is that it preserves the semantics of all the valid products compatible with ρ. This, however, is not guaranteed for the other valid products. Given that later we use this operator to compare sets of products, it is essential to characterize how the behaviour of these products is actually modified.

Theorem 5.9
Let fts be an FTS, $\rho : F \to \{\top, \bot\}$, and $p \in [\![fm]\!]$ such that p is not compatible with ρ. Then, ${fts^{[\rho]}}_{|p} = {fts}_{|p'}$ where p' is such that

$$f \in p' \Leftrightarrow \begin{cases} \rho(f), & f \in dom(\rho); \\ f \in p, & otherwise. \end{cases}$$

Proof. *Similarly with the proof of Theorem 5.8, we demonstrate that the transition sets of the two FTS are equal. To that aim, we show that*

$$(\gamma^{[\rho]})^{[p]} \Leftrightarrow \gamma^{[p']}$$

for any feature expression γ. Let f be a feature occurring in γ. If $f \in dom(\rho)$ then f is replaced by $\rho(f)$ in $\gamma^{[\rho]}$, $(\gamma^{[\rho]})^{[p]}$, and $\gamma^{[p']}$. Otherwise, f is replaced by $p(f)$ in $\gamma^{[p]}$, $(\gamma^{[\rho]})^{[p]}$, and $\gamma^{[p']}$. □

Intuitively, this result implies that in an FTS restricted to ρ, all the products behave as if they have every feature f such that $\rho(f) = \top$ and no feature f' such that $\rho(f') = \bot$. Features not determined by ρ take their value from p. Leaning on the two theorems above, we introduce our method to determine if a feature is conservative or regulative.

Theorem 5.10
Let fts be an FTS and f be a feature. Then f is conservative for a product $p \in [\![fm]\!]$ if and only if

$$p \cup \{f\} \in [\![fm]\!] \Rightarrow (fts \preccurlyeq_{FTS} fts^{[\{(f,\top)\}]})(p).$$

Similarly, f is regulative for p if and only if

$$p \cup \{f\} \in [\![fm]\!] \Rightarrow (fts^{[\{(f,\top)\}]} \preccurlyeq_{FTS} fts)(p).$$

PROOF. *We prove only the result for conservative features, the regulative case being similar. By definition of F-simulation we have*

$$(fts \preccurlyeq_{FTS} fts^{[\{(f,\top)\}]})(p) \Leftrightarrow fts_{|p} \preccurlyeq_{TS} fts^{[\{(f,\top)\}]}{}_{|p}.$$

for any $p \in [\![fm]\!]$. Let f be a feature. If $f \in p$, then

$$fts^{[\{(f,\top)\}]}_{|p} = fts_{|p} = fts_{|p \cup \{f\}}$$

by Theorem 5.8. Otherwise, we have $fts^{[\{(f,\top)\}]}_{|p} = fts_{|p\cup\{f\}}$ (see Theorem 5.9). This concludes the proof. □

By performing the algorithm that computes F-simulation, we can thus determine if a feature is conservative or regulative. However, this algorithm is intended for *any* couples of FTS defined over the same set of products. Taking into account that we check the simulation between an FTS and its restricted form may lead to heuristics that drastically reduce the computation time of the simulation relation. For example, $fts \preccurlyeq_{FTS} fts^{[\{f,\top\}]}$ immediately holds if there is no transition that excludes f itself, a feature required by f, or that requires a feature incompatible with f. However, we did not pursue the exploration of such heuristics.

5.4.3 Satisfiability Results

Now that we are able to identify if a new feature is conservative or regulative, we formally prove that we can avoid checking a subset of the new products introduced by the feature. This result is based on the preservation properties of F-simulation. Since we make use of the restriction operator to characterize the behavioural impact of a feature, we must determine under which conditions an FTS and one of its restricted form satisfy the same set of fLTL formulae.

Theorem 5.11
Let fts be an FTS and $\rho = \{f, \top\}$. Let $\psi = [\chi]\phi$ be an fLTL property. If $\chi \Rightarrow f$ then we have that

$$fts^{[\rho]} \models \psi \Leftrightarrow fts \models \psi.$$

PROOF. *Let p be a product. If $\chi(p)$ does not hold then p trivially satisfies $[\chi]\phi$ in both FTS. Otherwise, we have $f \in p$ since $\chi \Rightarrow f$. Hence, p is compatible with ρ. According to Theorem 5.8, $fts_{|p}$ and $fts^{[\rho]}{}_{|p}$ are syntactically equivalent and therefore both either satisfy of violate $[\chi]\phi$.* □

At last, we can formally establish which newly created products have not to be model checked after the addition of a feature. Let us consider the conservative case first. Intuitively, if a product violate a property then it still violates this property after adding a completely conservative feature to it. Therefore, when implementing a new conservative feature f in an SPL one has to verify only the products p equipped with the f such that $p \setminus \{f\}$ did not violate the property.

Theorem 5.12
Let fts be an FTS over FM fm with a set F of features. Let $f \notin F$ be a new feature. Let fts' be a second FTS over FM fm' such that $F' = F \cup \{f\}$ is its set of features, that f is completely conservative in fts', and that

$$\forall p \in [\![fm']\!] \bullet f \in p \vee p \in [\![fm]\!]$$
$$\forall p \in [\![fm]\!] \bullet [\![fts]\!](p) = [\![fts']\!](p)$$

Then for any fLTL formula $\psi = [\chi]\phi$ it holds that:

$$fts' \models \psi = (fts \models \psi) \wedge (fts' \models [\chi \wedge (fts \models \psi) \wedge f]\phi) \tag{5.7}$$

PROOF. *Let us consider the first clause in the right-hand side of Equation 5.7. The feature expression $(fts \models \psi)$ encodes the products that satisfied ψ in fts before f was added. Since f does not occur in $(fts \models \psi)$, the feature expression also encompasses products having f whose counterpart without f satisfied the formula. The other products having f cannot satisfy the formula since f is conservative.*

The second clause in the right-hand side of Equation 5.7 expresses the need to verify the products that satisfy one of the following:

1. χ: *the others trivially satisfy* ψ;
2. $fts \models \psi$: *the others violate* ψ *(see above);*
3. f: *those without the new feature were verified before.*

Equation 5.7 thus considers all products in $[\![fm']\!]$ *and specifies which of them violate the property.* □

A similar result holds in the regulative case. The proof is very similar and therefore omitted.

Theorem 5.13
Let fts *be an FTS over FM* fm *with a set* F *of features. Let* $f \notin F$ *be a new feature. Let* fts' *be a second FTS over an FM* fm' *such that* $F' = F \cup \{f\}$ *is its set of features, that* f *is completely regulative in* fts'*, and that*

$$\forall p \in [\![fm']\!] \bullet f \in p \vee p \in [\![fm]\!]$$
$$\forall p \in [\![fm]\!] \bullet [\![fts]\!](p) = [\![fts']\!](p)$$

Then for any fLTL formula $\psi = [\chi]\phi$ *it holds that:*

$$(fts' \models \psi) = (fts \models \psi) \vee (f \wedge (fts' \models [\chi \wedge (fts \not\models \psi) \wedge f]\phi))$$

Let us now consider the more general case in which f is conservative (respectively, regulative) only for some products. In this case, the products for which f is not conservative (respectively, regulative) must be checked anew after f is added to them. Leaning on this principle, we generalize the above two theorems to features that are conservative and regulative for some products. Again, we omit the proof since it follows from the proof of the two previous theorems.

Theorem 5.14
Let fts *be an FTS over FM* fm *with a set* F *of features. Let* $f \notin F$ *be a new feature. Let* fts' *be a second FTS over FM* fm' *such that* $F' = F \cup \{f\}$ *is its set of features, that* C *and* R *denote the feature expression encoding the products for which* f *is conservative and regulative in* fts'*, respectively, and that*

$$\forall p \in [\![fm']\!] \bullet f \in p \vee p \in [\![fm]\!]$$
$$\forall p \in [\![fm]\!] \bullet [\![fts]\!](p) = [\![fts']\!](p)$$

Then for any fLTL formula $\psi = [\chi]\phi$ it holds that:

$$(fts' \models \psi) = \big(((fts \models \psi) \wedge R) \vee (\neg f \wedge (fts \not\models \psi) \wedge C) \vee (f \wedge \neg(C \wedge fts \not\models \psi) \wedge (fts' \models [\chi \wedge \neg((fts \models \psi) \wedge R) \wedge \neg((fts \not\models \psi) \wedge C) \wedge f]\phi))\big).$$

This theorem allows one to minimize the number of new products to verify. It anticipates which new products will necessarily satisfy or violate the property.

5.5 Perspectives

F-simulation adds a significant milestone to the SPL verification theory, being at the center of advanced behavioural analyses such as abstraction, behavioural comparison, compositional reasoning, and more. The study of simulation quotients for FTS results in several simulation-based abstraction methods. However, our experiments suggest that their application for space reduction only yields marginal efficiency gains. To obtain more substantial improvements, our approach should be extended with other abstraction methods.

Apart from abstraction, there are many other uses of F-simulation. For instance, simulation-based verification allows one to verify properties modelled as automata. There are also SPL-specific uses of our theory. For instance, we can formally characterise the behavioural impact of features in an SPL by analysing its conservative and regulative capabilities. Inter-SPL comparison is also possible. If we consider two SPLs having an equal set of valid products, the behavioural inconsistencies between them can be highlighted and presented in the form of an automata (*viz.* an FTS). A similar approach for MTS is studied by Sassolas *et al.* [153].

More generally, this contribution is part of a bigger project meant to combat the state explosion problem in FTS model checking. Although several state-space reduction heuristics have helped to improve the efficiency of single-system model checkers, those cannot be applied to FTS as they are. Because of the presence of variability inside the behavioural model (*viz.* FTS), undesired behaviour may be created because of a careless application of those optimisations. Thus, we must first adapt them specifically for FTS. Also, the worst-case time complexity of the FTS model checking algorithms is exponential in the number of features, as proven in [58]. We believe that under certain conditions, we can design a new algorithm whose complexity is *linear* in the number of features. In the next chapter, we study counter-example guided abstraction refinement [48], which looks particularly

promising: a coarse abstraction is rapidly and automatically computed and is then refined iteratively using the false negatives found during verification.

Compositional reasoning appears as another promising approach to this end. It permits to achieve separation of concerns at the level of the features, thereby allowing one to estimate the impact of a feature or modifications applied to a feature. Furthermore, the evolution problem for SPL can be reduced to the modular analysis of features and thus be solved thanks to the same techniques. This, however, remains a difficult problem mainly because features can be arbitrarily invasive. Moreover, conflicts and interactions between them often make a modular verification impossible. These challenges highlight the need for identifying which classes of features can be considered in isolation. The definition and detection of conservative and regulative features are but the first steps forward. Still we believe that they are solid bricks on which compositional methods can be build on. Finally, optimising the verification of only the two aforementioned types of features might lead to little improvement if these appear rarely. In the case studies we use for experiments further in this thesis, these classes of features never occur. This has yet to be confirmed by evaluating their occurrence rate in real-world SPLs.

Chapter 6

Counterexample Guided Abstraction Refinement

Although SPL model-checking methods already bring order-of-magnitude improvements in verification time as opposed to enumerative approaches, they are still confronted to a two-source, exponential blow-up. First, they suffer from the state-explosion problem inherent to model checking. Second, their practical time complexity still grows more than linearly with the number of features [53]. Model abstraction, which creates more concise models of the system, is one of the most effective answers to state explosion. However, this reduced size often comes at the cost of inaccuracies in the models, and thereby affects the properties they satisfy. A reported counterexample can indeed be *spurious*, that is, it exists within the abstract model but not in the real, concrete model. In this case, the abstraction must be refined to eliminate this false positive. Common methods to perform this refinement make use of the spurious counterexample itself. They give rise to Counterexample Guided Abstraction Refinement (CEGAR), i.e. abstraction techniques that iteratively refine an abstract model until either they find a real counterexample or they can prove the absence of violation [48]. The starting step of CEGAR is to build a first abstraction of the model. To that aim, an existential abstraction is commonly used.

Definition 6.1 (Existential Abstraction)
[48] *An existential abstraction h is a surjection $h : S \to \widehat{S}$ such that $h(s) = h(s') \Rightarrow L(s) = L(s')$. An abstraction of a TS $ts = (S, Act, Trans, I, AP, L)$ under h is a TS $ts_h = (\widehat{S}, Act, Trans_h, I_h, AP, L_h)$ where $\widehat{S} = \{h(s) | s \in S\}$, $I_h = \{h(s_0) | s_0 \in I\}$, $L_h(h(s)) = L(s)$, and $trans_h$ is defined such that $s \xrightarrow{\alpha} s' \Leftrightarrow h(s) \xrightarrow{\alpha} h(s')$.*

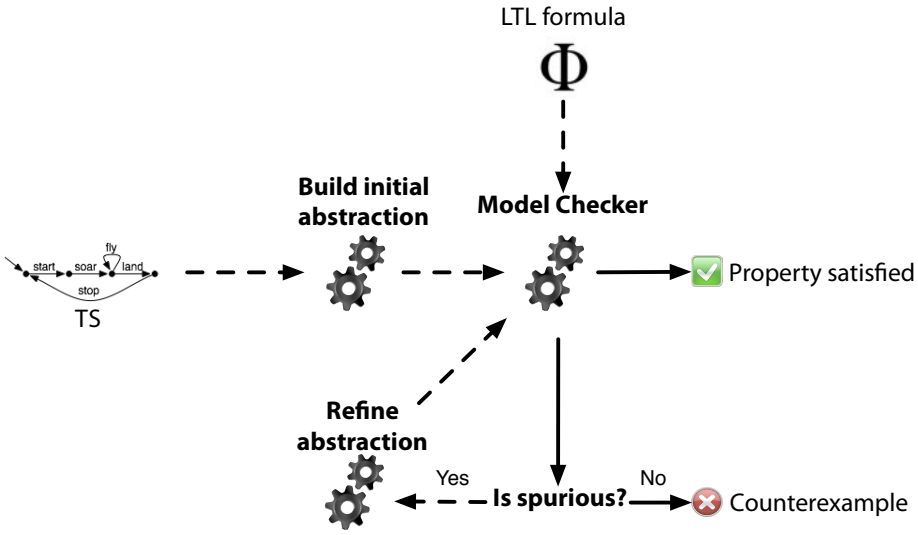


Figure 6.1: Overview of the CEGAR procedure

Existential abstractions can facilitate verification thanks to two interesting properties. First, ts_h is such that $ts \preccurlyeq_{TS} ts_h$ [48]. Second, they can be defined such that $\widehat{S}$ is significantly smaller than S. Therefore, for a given LTL formula ϕ we can prove that the concrete TS satisfies ϕ by proving that ts_h satisfies ϕ. However, if ts_h does not satisfy the formula then ts may still satisfy ϕ; it means that the reported counterexample is *spurious*. This may occur when ts_h has strictly more traces than ts. In this case, it is required to *refine* the abstraction in order to eliminate the spurious counterexample. From the refinement, we obtain a new abstraction function h' which is less coarse than h, that is, $ts \preccurlyeq_{TS} ts_{h'} \preccurlyeq_{TS} ts_h$.

CEGAR is an established refinement method that consists in using a spurious counterexample to refine the abstraction [48]. Figure 6.1 illustrates the CEGAR process. From the concrete TS, we build an initial abstraction based on a given existential abstraction function. We then feed the abstract model into the model checker together with the formula to check. If the abstraction satisfies the formula, then so does the concrete TS. Otherwise, we replay the counterexample on the concrete system to determine whether it is spurious. If it is not, then the concrete TS violates the formula; else we refine the abstraction on the basis of the counterexample and we repeat the process.

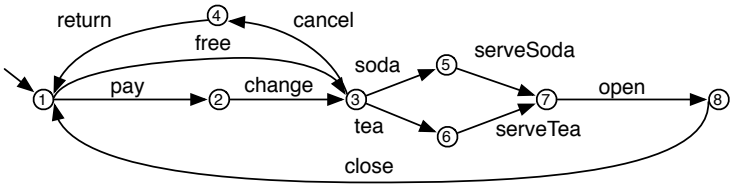
In spite of their success in single-system model checking, abstraction techniques have not yet been studied in the context of SPLs. In this chapter, we fill this gap and propose SPL-specific abstraction procedures based on

CEGAR. Applying CEGAR to SPLs is more tedious because a counterexample can be real for some products and spurious for others. This observation leads us to two refinement strategies: one refines the model as soon as it finds a spurious counterexample, whereas the other performs the spuriousity check and the refinement after the discovery of all the counterexamples. As for the abstraction of the model, we distinguish between (1) *feature abstraction* that modifies only the variability information contained in the model, (2) *state abstraction* that only merge states as in single-model abstraction, and (3) *mixed abstraction* that combines the previous two types. This latter type is the most complicated to implement, as spuriousness can originate from the merging of states, the abstraction of features, or both. Throughout the paper, we systematically prove the correctness of our approach on the basis of F-simulation. We implemented both abstractions and their combination in ProVeLines, an SPL model checker we developed (see more in Chapter 9). We carried out experiments to evaluate the efficiency of different combinations of refinement strategies and abstractions. Our results tend to show that when combined with the appropriate refinement strategy, state abstraction brings performance gains on average, although it yields longer verification time in a minority of cases. As for feature abstraction, it generally results in small losses of performance but can achieve huge decreases of verification time in some cases, especially when there is a lot commonality between products or when the formula is satisfied by all products. Preliminary experiments on mixed abstraction tend to show that its performance is comparable to that of state abstraction, although slightly worse on average. Other abstractions of this kind could, however, be designed as part of future work and yield better results.

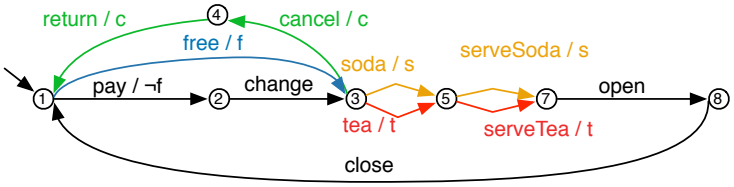
6.1 CEGAR Strategies in SPL Model Checking

State-of-the-art abstraction techniques cannot be reused as such in SPL verification. There are two fundamental differences between single-system and SPL model checking. First, a model for *multiple* products, *viz.* an FTS, is checked as opposed to a single-system model, *viz.* a TS. This implies that (1) an appropriate abstraction function must preserve the behaviour of all the products modelled by the FTS, and (2) this function can modify the labelling function γ. To produce FTS abstractions, we can thus either merge states as in single-system abstraction, weaken feature expressions to make transitions available to more products, or both. These three solutions are respectively called *feature abstraction*, *state abstraction*, and *mixed abstraction*.

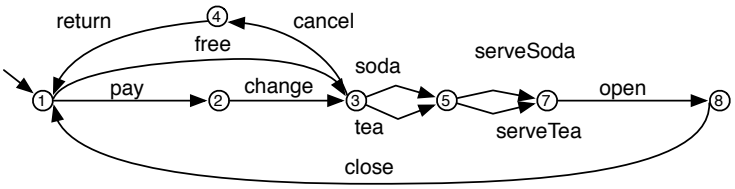
Example 6.1.1 *Figure 6.2 illustrates our three types of FTS abstraction applied to the vending machine FTS (see Figure 3.2). In Figure 6.2a, all transitions are made available to all products, thereby creating additional*



(a) Feature abstraction



(b) State abstraction



(c) Mixed abstraction

Figure 6.2: Examples of the three forms of abstraction.

behaviour for each of them. In Figure 6.2b, the feature expressions have not changed, but state 5 and state 6 have been merged. This means that if the vending machine can serve tea and soda, when the customer orders a tea, the machine may deliver a soda instead. Finally, the abstraction shown in Figure 6.2c combines the previous two abstractions. Consequently, it makes the aforementioned inconsistent behaviour available to all valid products.

A second requirement is that the CEGAR process cannot stop after only one real counterexample is found if this counterexample is not executable by all products. If it does stop, the model checker could ignore violations performed by other products; the SPL model-checking problem would thus not be answered appropriately. To address these requirements, we extend the definition of existential abstraction and we provide property preservation proofs. Then we present two SPL-specific CEGAR procedures that can verify all the products.

As before, we consider existential abstraction functions as these guarantee behaviour preservation. To transpose this concept to FTS while maintaining the preservation property, we add another requirement to the abstraction function: any product that can execute a concrete transition must

be able to execute the corresponding abstract transition. This leads to the following new definition of existential abstraction.

Definition 6.2 (F-abstraction)
An F-abstraction is a surjection $h : S \to \widehat{S}$ *such that* $h(s) = h(s') \Rightarrow L(s) = L(s')$. *An abstraction of a FTS* fts *under* h *is an FTS* $fts_h = (\widehat{S}, Act, Trans_h, I_h, AP, L_h, d, \gamma_h)$ *where* $\widehat{S}$, Act, $Trans_h$, I_h, AP, L_h *are as in Definition 6.1 and* γ_h *is such that*

$$\Big(\bigvee_{s \in h^{-1}(\widehat{s}), s' \in h^{-1}(\widehat{s'}) \bullet s \xrightarrow{\alpha} s'} \gamma(s, \alpha, s') \Big) \Rightarrow \gamma_h(\widehat{s}, \alpha, \widehat{s'}) \tag{6.1}$$

with $h^{-1}(\widehat{s}) = \{s \in S \mid h(s) = \widehat{s}\}$.

Intuitively, h is responsible for state abstractions, whereas γ_h creates feature abstractions. Equation 6.1 specifies that an abstract transition must be executable by all the products that can execute at least one of the corresponding concrete transition. Any FTS obtained from an F-abstraction completely simulates the concrete FTS for all products.

Theorem 6.1
Let fts *be an FTS and* h *be an F-abstraction function. Then* $(fts \preccurlyeq_{FTS} fts_h) = fm$.

Proof. *Since existential abstractions from TS preserve the behaviour of the concrete TS modulo simulation [48], any given transition of* fts *is also a transition of* fts_h. *It thus remains to prove that this transition is available in* fts_h *for at least the same set of products, that is, for any states* s, s' *we have*

$$(s, \alpha, s') \in Trans_{|p} \Rightarrow (\widehat{s}, \alpha, \widehat{s'}) \in (Trans_h)_{|p}$$

where $Trans_{|p}$ *and* $(Trans_h)_{|p}$ *denote the transitions of* $fts_{|p}$ *and* $(fts_h)_{|p}$*, respectively. This property directly follows from the definition of* γ_h*:*

$$
\begin{aligned}
&\Big(\bigvee_{s,s' \bullet s \xrightarrow{\alpha} s'} \gamma(s,\alpha,s')\Big) \Rightarrow \gamma_h(\widehat{s},\alpha,\widehat{s'}) \\
\Leftrightarrow &\bigwedge_{s,s' \bullet s \xrightarrow{\alpha} s'} \gamma(s,\alpha,s') \Rightarrow \gamma_h(\widehat{s},\alpha,\widehat{s'}) \\
\Rightarrow &\bigwedge_{s,s' \bullet s \xrightarrow{\alpha} s'} \gamma(s,\alpha,s')(p) \Rightarrow \gamma_h(\widehat{s},\alpha,\widehat{s'})(p) \\
\Leftrightarrow &\forall s, s' \bullet (s,\alpha,s') \in Trans_{|p} \Rightarrow (\widehat{s},\alpha,\widehat{s'}) \in (Trans_h)_{|p}
\end{aligned}
$$

with $s \in h^{-1}(\widehat{s})$ *and* $s' \in h^{-1}(\widehat{s'})$*. Hence* $(fts \preccurlyeq_{FTS} fts_h) = fm$*.* □

We are now ready to present our CEGAR procedures for FTS. Given that an FTS model checker may return several counterexamples, two refinement strategies can be followed. The first strategy, called *Find All Before Refining (FABR)* consists in waiting for the model checker to find all the counterexamples it should return, then checking the spuriousity of each of them, and refining the model if need be. Conversely, the second strategy, *Refine When Found One (RWFO)*, checks spuriousity and possibly refines the model as soon as the model checker finds one counterexample.

Figure 6.3 gives an overview of FABR. First, we apply an initial F-abstraction h to obtain an abstract FTS which is given to the model checker along with the FM and an fLTL formula $[\chi]\Phi$. The initial abstraction is built according to a chosen abstraction technique. The model checker returns (1) the set of products satisfying the formula, and (2) the set of products violating the formula together with counterexamples. We then determine which counterexamples are spurious. For those that are not spurious, we record the products able to execute it as violating the formula. If there exist spurious counterexamples, we successively refine the abstraction based on these counterexamples to build a new F-abstraction h' applied to the same set of states, and produce an FTS $fts_{h'}$ such that

$$(fts_{h'} \preccurlyeq_{FTS} fts_h) = fm \tag{6.2a}$$
$$(fts \preccurlyeq_{FTS} fts_{h'}) = fm \tag{6.2b}$$
$$[\![fts_h \preccurlyeq_{FTS} fts_{h'}]\!] \subset [\![fm]\!] \tag{6.2c}$$

and we verify $fts_{h'}$ against the property. During this new search, we can safely ignore the products that were previously recognized as satisfying. Indeed, by definition of F-simulation any product that satisfies $[\chi]\Phi$ in fts_h also satisfies the formula in $fts_{h'}$ and fts. We can also ignore products that can execute a *real* counterexample, since the new check will report them as

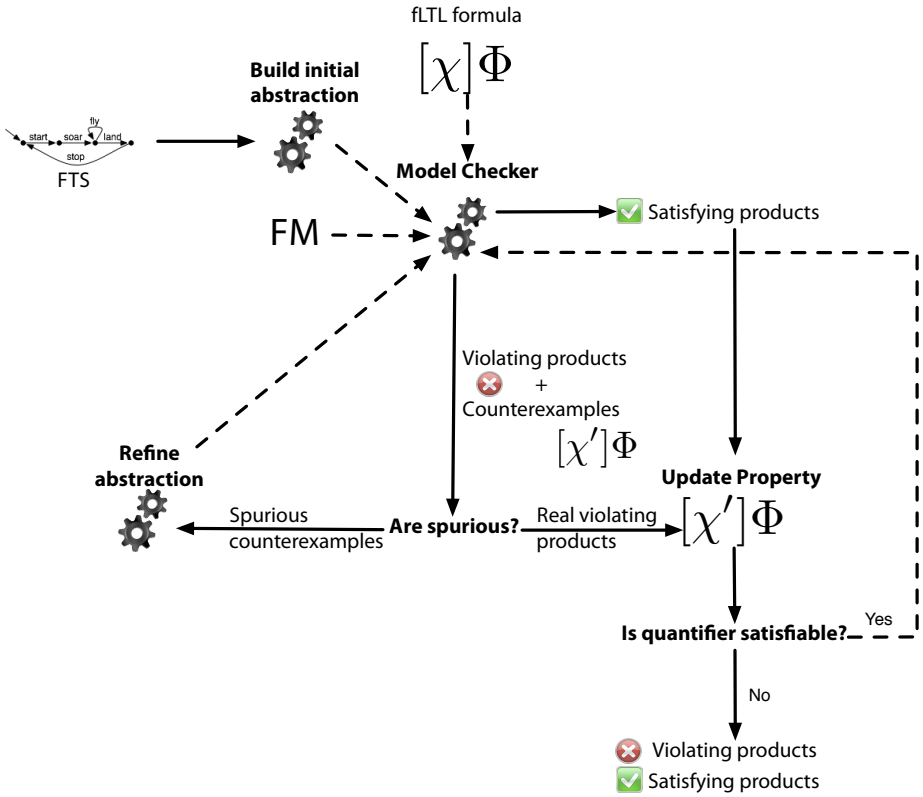


Figure 6.3: Overview of FABR

violating as well. To prevent the model checker to consider these products, it suffices to transform the feature quantifier χ into $\chi' = \chi \wedge \neg\upsilon$ where υ is the feature expression characterizing the products to ignore. We repeat the process until the model checker returns no spurious counterexample, or equivalently while the updated product quantifier χ' is satisfiable. As a result, we pinpoint the products that satisfy the formula and those that do not.

The RWFO strategy is illustrated in Figure 6.4. As before, we build an initial abstraction and check it. If all the products satisfy the formula, we stop the process since it means that the products also satisfy the formula in the original FTS. Otherwise, we interrupt the verification as soon as the model checker finds a counterexample. If this counterexample is spurious, we refine the abstraction and start the model checking again. Otherwise, we update the formula to ignore the products that can execute the counterexample. If the resulting product quantifier χ' is satisfiable, we *carry on the verification procedure from when the counterexample was found.* We repeat the process until there is no more counterexample or the product quantifier becomes unsatisfiable. In the latter case, it means that all the products

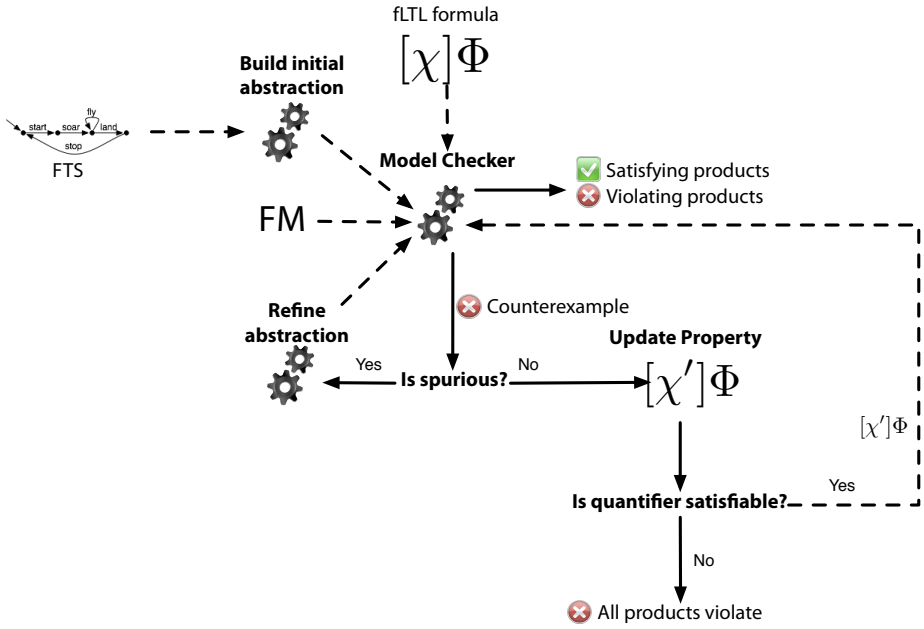


Figure 6.4: Overview of RWFO

violate the formula.

The following theorem establishes the termination and the correctness of our CEGAR strategies.

Theorem 6.2
Let a refinement procedure that yields abstractions such that Equations 6.2 hold. Then, the CEGAR strategies terminate, are sound, and are complete.

PROOF. (Termination) *At the end of a verification, the CEGAR procedures either terminate, update the product quantifier χ into χ', or trigger a refinement. Given that $[\![\chi']\!] \subset [\![\chi]\!]$ the number of updates is finite. By Equations 6.2, the number of refinements is also finite.*

(Correctness) *Let p be a valid product. After a verification, three cases may occur:*

1. p is reported as satisfying the formula. By Definitions 5.2 and 6.2, p satisfies the formula in the concrete FTS as well.

2. p is associated to a real counterexample, and thus violates the formula in the concrete FTS as well.

3. p is associated to spurious counterexamples only. In this case, p will be checked again after the refinement.

Furthermore, the definition of χ' guarantees that only products that are known to satisfy or violate the formula are ignored in upcoming verifications. □

The strategies are independent of the actual abstraction and refinement functions being used, as long as these satisfy Definition 6.2 and Equations 6.2. In the following section, we show how to actually build different types of abstraction from a concrete model.

6.2 Building FTS Abstraction

Existing abstraction techniques generally do not work directly on TS. Instead, they rely on higher-level representations that include an explicit notion of variable [48]. These are commonly named *program graphs* [16]. In the context of SPLs, the products will be represented by a set of programs that share commonalities. In order to specify all these programs in a single compact model, one can borrow the concept of feature expression introduced in FTS and apply it to program graphs [53].

Definition 6.3 (Featured Program Graph)
Let $V = \{v_1, \ldots, v_n\}$ be a set of variables, $\tau : V \to Types$ a type function, $Pred(V)$ the set of predicates over V, $Asgn(V)$ the set of assignments over V of the form $v := expr$ where $expr$ is an expression, and $Eval(V) = \tau(v_1) \times \cdots \times \tau(v_n)$ the set of variable valuations. A Featured Program Graph (FPG) is a tuple $(Loc, V, \tau, Act, loc_0, init, Trans, fm, \gamma)$ where

- *Loc is a set of locations;*
- *$Act = Pred(V) \cup Asgn(V)$ is the set of actions;*
- *loc_0 is the initial location;*
- *$Init \subseteq Eval(V)$ are the set of possible initial valuations;*
- *$Trans \subseteq Loc \times Act \times Loc$ is the transition relation;*
- *fm is an FM over a set F of features;*
- *$\gamma : Trans \to 2^{(2^F)}$ associates each transition with a feature expression.*

One can define the semantics of an FPG in terms of FTS (see [53] for details). Intuitively, given an FPG the state space of the underlying FTS

is $Loc \times Eval(V)$, that is, an FTS state is defined as a location and a variable valuation. The set of initial states is $Loc_0 \times Init$. Transitions are determined according to the variable values of the source state, as well as to the predicates and assignments that label the transitions of the current FPG location. Formally, the transition relation is the smallest relation satisfying

$$\frac{l \xrightarrow{p} l' \wedge v \models p}{(l, v) \xrightarrow{p} (l', v)}$$

$$\frac{l \xrightarrow{x:=expr} l' \wedge v' = [x := expr]v}{(l, v) \xrightarrow{x:=expr} (l', v')}$$

where $[x := expr]v$ is v after assigning to x the value to which $expr$ is evaluated, and $p \in Pred(V)$. Given a formula to check, the set AP of atomic propositions is the set of predicates over V that occur in the formula. Feature expressions are directly obtained from γ. In what follows, we interchangeably use γ to denote the transition labelling function of both an FPG and its underlying FTS. We now present two families of methods to build and refine abstractions of FPG, and discuss their implications on the CEGAR strategies presented above.

6.2.1 Feature Abstraction

The first abstraction type we propose consists in abstracting from the variability between the products. Concretely, we replace each feature expression $\gamma(t)$ labelling a transition t by another feature expression $\gamma_h(t)$ such that $\gamma(t) \Rightarrow \gamma_h(t)$. The set of states and transitions are, however, left untouched. The abstraction function is thus defined as $h : S \to S : h(s) = s$. This transformation preserves the behaviour of every product, by Definition 6.2 and Theorem 6.1.

For the initial abstraction, we propose to completely abstract all the feature expressions of the FPG, that is, we replace each of them by $\top$. This comes down to associating to every product of the underlying FTS (S, Act, $Trans$, I, AP, L, d, γ) the semantics of the TS (S, Act, $Trans$, I, AP, L). An undeniable advantage of this abstraction is that single-system model checking algorithms can perform the verification. If the initial abstraction satisfies a given formula then all the products satisfy it. Otherwise, the model checker returns a counterexample. Given that the F-abstraction did not modify the state space, the counterexample is necessarily an existing execution in the aforementioned TS. It may happen that no product can execute it (due to incompatible feature expressions), though; indeed, the TS semantics subsumes the union of the semantics of the projection of the FTS onto all valid products [59]. For instance, an execution that triggers transitions t and t', with $\gamma(t) = \neg\gamma(t')$ would be executed in the abstract FTS but not in the concrete one. Even if at least one product can execute

the counterexample, it does not mean that all the products can. More generally, if b_h (respectively b) is the feature expression characterizing the set of products that can execute the counterexample in FTS_h (respectively FTS), then the counterexample is spurious for products satisfying $b \wedge \neg b_h \wedge \chi$.

If the counterexample is spurious for at least one product, the abstraction must be refined. Here, the refinement consists in replacing the abstract feature expression of each FPG transition that is executed during the counterexample by its concrete feature expression. This implies that for each FTS transition t, the feature expression labelling t, noted $\gamma_{h'}(t)$, is such that $\gamma_{h'}(t) \Rightarrow \gamma_h(t)$. Hence, the resulting abstraction function h' guarantees that

$$(FTS_{h'} \preccurlyeq_{FTS} FTS_h) = fm$$
$$(FTS \preccurlyeq_{FTS} FTS_{h'}) = fm$$

where h is the initial F-abstraction. Since we updated the feature expression of at least one transition that impacts the behaviour, it follows that $[\![FTS_h \preccurlyeq_{FTS} FTS_{h'}]\!] \subset [\![fm]\!]$. Equations 6.2 are thus satisfied, which guarantees the termination and the correctness of the aforementioned CEGAR procedures.

6.2.2 State Abstraction

Unlike the first one, the second abstraction type relies on transposing the principles of classical abstraction functions to SPL CEGAR. More precisely, we consider *predicate abstraction*, one of the most applied methods for program abstraction [98, 48, 49]. Several model checkers make use of predicate abstraction to speed up the verification. At the last TACAS software verification competition [26], UFO [4], LLBMC [87], CPAChecker [27], and ESBMC [146] were nominated the fastest model checkers for product lines. Unlike ours, these tools do not treat features as first-class citizens. This means that either they rely on a product-by-product approach or their analysis procedure consists in verifying a model that includes the behaviour of all products (i.e. based on a product simulator [12] or on configuration lifting [150]). In the first case, abstraction can be applied to the models of the individual products but the commonality between these is not exploited. The second case offers interesting perspectives regarding an efficient application of abstraction on all products. However, this approach can determine whether or not *all* products satisfy the property, while we want to identify *which* products are erroneous.

Let $Pred = \{p_1, \ldots, p_n\} \subseteq Pred(V)$ be the set of predicates that occur in the FPG or in the formula to check. Let $h : (Loc \times Eval(V)) \rightarrow (Loc \times 2^{Pred})$ be an existential abstraction function such that $h(l, v) = (l, \{p_i \in Pred \mid v \models p_i\})$. Intuitively, FDG locations are preserved in abstract states but

a variable valuation is abstracted by the subset of predicates it satisfies. Since the predicates of the formula to check are included in $Pred$ and thus contribute to the definition of h, we have $h(s) = h(s') \Rightarrow L(s) = L(s')$ and $L(h(s)) = L(s)$. The abstract transition relation $Trans_h$ is defined as the smallest relation satisfying:

$$\frac{l \xrightarrow{p} l' \wedge p \in P}{(l, P) \xrightarrow{p} (l', P)}$$

$$\frac{l \xrightarrow{x:=expr} l' \wedge ([P] \wedge wp(x := expr, [P']) \not\models \bot)}{(l, P) \xrightarrow{x:=expr} (l', P')}$$

where $[P] = \bigwedge_{p \in P} p \wedge \bigwedge_{q \in Pred \setminus P} \neg q$ and $wp(x := expr, \phi)$ is ϕ with each occurrence of x replaced by $expr$. Intuitively, P' is the set of predicates that are satisfiable after assigning $expr$ to x given that every predicate in P did hold before the assignment and that every predicate not in P did not hold. In this abstraction, we do not change the feature expressions, that is, $\gamma_h(t) = \gamma(t)$ for any transition t. We thus have $(FTS \preccurlyeq_{FTS} FTS_h) = fm$.

Technically, the definition of state abstraction is very similar to state-of-the-art predicate abstractions. The difference rather lies in the detection and the analysis of counterexamples, which now depends on variability. Indeed, the notion of spuriousness is more subtle in this case. First, a spurious counterexample may not correspond to an existing sequence of transitions in the concrete FTS. Second, it can be an existing execution but only for a subset of the products. We thus propose a method to detect spurious counterexamples that can be represented as finite sequences. This method is inspired by the SplitPATH algorithm used in single-system CEGAR [48]. It consists in identifying a set of states whose merging has yielded a spurious counterexample, and then splitting this set so as to make the counterexample disappear. We can also extend our technique with support for any kind of counterexample using unwinding techniques along the lines of Clarke *et al.* [48]. The key idea of our method is to remember, while replaying the counterexample, for which products we can execute each step of the replay. Formally, let $\sigma = h(s_0)\alpha_0 \dots h(s_m)$ be an abstract counterexample and σ_b the feature expression characterizing the products able to execute it. Let $S_0 = \{(i, \chi) \mid i \in I\}$ and

$$S_i = \{(s, b) \in h^{-1}(h(s_i)) \times 2^{(2^F)} \mid (s', b') \in S_{i-1} \wedge \big(b = (b' \wedge \bigvee_{s' \xrightarrow{\alpha_{i-1}} s} \gamma(s', \alpha_{i-1}, s))\big)\}.$$

Intuitively, if $(s, b) \in S_i$ then s can be reached by products satisfying b by replaying the first i steps of the counterexample. If for a product p there does

not exist $(s, b) \in S_i$ with $b(p)$ then p cannot execute the counterexample. Thus σ is spurious for products

$$\{p \in [\![fm]\!] \mid \sigma_b(p) \wedge \nexists (s, b) \in S_m \bullet b(p)\}$$

or equivalently those satisfying

$$fm \wedge \sigma_b \wedge \neg(\bigvee_{(s,b)\in S_m} b).$$

We also name these products spurious. The correctness of this detection procedure is derived from the proof of SplitPATH [48] and the definition of projection.

To refine the abstraction, we first have to identify the execution steps during which products are discovered not to be able to execute the counterexample. Let $\sigma_{b'}(i)$, $0 \leq i \leq n$ represent the products that can execute the counterexample in the concrete FTS up to step i, with $\sigma_{b'}(0) = \chi$. Then for any $i > 0$ the feature expression $\sigma_b \wedge \neg\sigma_{b'}(i)$ characterizes the products that are spurious at step i, and $\sigma_b \wedge \sigma_{b'}(i) \wedge \neg\sigma_{b'}(i+1)$ represents the products that became spurious exactly at step $i + 1$. If the latter feature expression is satisfiable, a refinement should be performed at that step.

In predicate abstraction, a refinement consists in adding a new predicate in the construction of the abstraction function. An abstract state will therefore contain more information and the abstraction function will produce finer abstractions. The predicate to add can be determined by means of Craig interpolation [72]. Given two formulae ϕ and ψ with $\phi \wedge \psi \models \bot$, a Craig interpolant is a formula ϕ' written using the intersection of the vocabularies of ϕ, ψ such that $\phi \Rightarrow \phi'$ and $\phi' \wedge \psi \models \bot$. Loosely speaking, ϕ' can be seen as an explanation of why $\phi \wedge \psi$ is not satisfiable. In our case, ϕ is the conjunction of the source state's predicates and the transition predicate (if any), ψ is the conjunction of the target state's predicates, and ϕ' is the conjunction of ϕ with the predicate to add as part of the refinement.

The occurrence of a spurious product means that the abstraction has created a transition between a state $s_i \in S_i$ and a state $s_{i+1} \in S_{i+1}$ that does not actually exist. To eliminate this spurious counterexample, the new abstraction has to make the distinction between s_i and the states in S_i that actually have a transition to s_{i+1}. We thus compute a Craig interpolant between $\bigwedge_{v\in V} v = eval(s_i, v)$ and $\bigvee_{(s_j \bullet s_j \xrightarrow{\alpha} s_{i+1})} \bigwedge_{v\in V} v = eval(s_j, v)$. This yields the new predicate to add as a refinement to the abstraction. Once the refined abstraction is obtained, we perform a new verification limited to the spurious products.

Let h' be the refined abstraction yielded by that method. Then $h'(s) = h'(s') \Rightarrow h(s) = h(s')$ and the abstract transition relation in $FTS_{h'}$ is finer than in FTS_h. Furthermore, the refinement method removes from at least one product the ability to execute the i-th step of the spurious counterexample. Hence Equations 6.2 hold.

6.2.3 Mixed Abstraction

Given that the above two abstraction methods do not modify the same constructs, we can straightforwardly combine them to form coarser abstractions. Thereby, we obtain an initial existential abstraction function h as defined in Subsection 6.2.2 together with an abstract labelling function γ_h as presented in Subsection 6.2.1. Although the construction of the abstraction is straightforward, the detection of spurious counterexamples and the refinement process become more complex. Indeed, spurious products may originate from different factors, *viz.* the abstraction of feature expressions, the predicate abstraction, or their combination. This raises the questions of how to associate spuriousness cases with their appropriate origin(s), and how to guide the refinement process in case of multiple origins.

As before, we compute sets of states S_i containing the states that can be reached upon the execution of the first i steps of the counterexample, together with the products able to reach them. The actual definition of S_i is as previously. However, since we have to take into account that feature expressions are abstracted as well, the detection algorithm should determine the reason why a product became spurious and refine either the labelling function or the abstraction of the state space.

Procedure 3 formalizes our detection and refinement method. It consists in replaying the counterexample step by step, and check at each step whether we discover spurious products (Lines 5–19). At each iteration, σ_b represents the products that violate the property in fts_h and have not been discovered to be spurious, whereas $\sigma_{b'}$ represents the products that can execute the counterexample in fts up to the current step. For a given step i, we first check whether there are new spurious products due to the abstraction of feature expression (Lines 6–12). To achieve that, we compute the feature expression that would label the abstract transition should the feature abstraction not have been applied, add it to $\sigma_{b'}$, and compare the result with σ_b. Condition at line 8 means that any product in $[\![\sigma_b]\!] \setminus [\![\sigma_{b'}]\!]$ is a spurious product. If there is at least one such product, a refinement of the feature abstraction function is applied. Accordingly, we modify the feature expression labelling the abstract transition. Next, we check whether predicate abstraction yielded additional spurious products at step i by using the method presented in Subsection 6.2.2 (Lines 13–18). If it did, we compute and register the interpolant that will act as a refinement of the current predicate abstraction (Lines 15–18). The algorithm ends up returning either true if no refinement was needed, or a new F-abstraction together with the set of really violating products (Lines 21–22). This F-abstraction is built according to the refinement procedures of the previous two abstractions.

Input: fts and fts_h such that $[\![fts \preccurlyeq_{FTS} fts_h]\!] = [\![fm]\!]$, a quantifier χ, $\sigma = \widehat{s_0}\alpha_0 \dots \widehat{s_m}$ and $\sigma_b \in 2^{(2^F)}$ such that $\sigma \in Prefix([\![(fts_h)_{|p}]\!]_{TS})$ for all $p \in [\![\sigma_b]\!]$.
Output: $True$ if $\sigma \in Prefix([\![(fts_h)_{|p}]\!]_{TS})$ for all $p \in [\![\chi \wedge \sigma_b]\!]$, $(h', \sigma_{b'})$ such that Equations 6.2 hold and $\sigma \in Prefix([\![(fts_h)_{|p}]\!]_{TS})$ for all $p \in [\![\chi \wedge \sigma_b]\!]$ otherwise.

$\sigma_{b'} \leftarrow \chi \wedge fm$;
$refined \leftarrow \bot$;
$Pred' \leftarrow Pred$;
$\gamma_{h'} \leftarrow \gamma_h$;
for $i = 1$ *to* m **do**
  $\gamma_{h'}(\widehat{s_{i-1}}, \alpha_{i-1}, \widehat{s_i}) \leftarrow \bigvee_{s_{i-1} \in h^{-1}(\widehat{s_{i-1}}), s_i \in h^{-1}(\widehat{s_i})} \gamma(s_{i-1}, \alpha_{i-1}, s_i)$;
  $\sigma_{b'} \leftarrow \sigma_{b'} \wedge \gamma_{h'}(\widehat{s_{i-1}}, \alpha_{i-1}, \widehat{s_i})$;
  if $\sigma_b \not\Rightarrow \sigma_{b'}$ **then**
    $\gamma_h(\widehat{s_{i-1}}, \alpha_{i-1}, \widehat{s_i}) \leftarrow \gamma_{h'}(\widehat{s_{i-1}}, \alpha_{i-1}, \widehat{s_i})$;
    $\sigma_b \leftarrow \sigma_b \wedge \sigma_{b'}$;
    $refined \leftarrow \top$;
  end
  $\sigma_{b'} \leftarrow \sigma_{b'} \wedge \bigvee_{(s,b) \in S_i} b$;
  if $\sigma_b \not\Rightarrow \sigma_{b'}$ **then**
    $Pred' \leftarrow Pred' \cup \{Craig(\widehat{s_{i-1}} \wedge \alpha_{i-1}, \widehat{s_i})\}$;
    $\sigma_b \leftarrow \sigma_b \wedge \sigma_{b'}$;
    $refined \leftarrow \top$;
  end
end
if $refined$ **then**
  $(h', \gamma_{h'}) \leftarrow abstract(fts, h, Pred', \gamma_{h'})$;
  return $(h', \gamma_{h'})$;
end
else
  return $\top$;
end

Algorithm 3: IsSpurious($fts, fts_h, \chi, \sigma, \sigma_b$)

6.3 Evaluation

In order to compare and evaluate the potential benefits of our CEGAR procedures and our F-abstractions, we implemented them on top of FTS model checking algorithms [59, 56]. In this section, we describe our implementation, present the results of experiments we carried out, and attempt to infer general cases where it is particularly rewarding to apply CEGAR instead of standard FTS algorithms.

ProVeLines is a product line of model checkers for SPLs we built. It allows one to verify the behaviour of an SPL modelled as an FTS against properties expressed in different logics. The variants of ProVeLines are semi-symbolic model checkers: they encode the SPL products symbolically as feature expressions, but explore the state space explicitly. Among the input languages accepted by ProVeLines one finds fPromela [53], a feature-aware extension of Promela [114]. fPromela provides constructs to declare variables and statements, as well as to label the latter with feature expressions. It can thus be regarded as a concrete syntax for featured program graphs, and is consequently appropriate as an input to our CEGAR procedures. We give more detailed information about ProVeLines in Chapter 9.

On the basis of our implementation, we carried out experiments to evaluate in which cases and to what extent our CEGAR-based verification methods are more efficient than a plain application of the FTS algorithms presented in our previous work [59, 56]. The benefits of CEGAR originate from two factors. First, the existential abstraction reduces the size of the state space, and thus the number of states to explore before discovering a counterexample. Second, the abstraction of feature expressions increases the commonality between the behaviour of the products and thereby augments the potency of SPL-specific optimizations such as late splitting (see Section 3.3). However, the approximation due to abstraction and the consequent needs for refinement may cancel these benefits, or even worsen the verification time. Indeed, after applying a refinement, a completely new verification must be performed on the refined model. After several refinements, the number of states explored by all the verifications altogether can exceed that of the standard algorithms. The performance of the CEGAR procedures thus depends on several factors including the topology of the model, the chosen F-abstractions, and the property.

As an attempt to estimate the impact of these factors, we computed the time needed by both CEGAR strategies combined with different F-abstractions and implemented on top of the FTS algorithms to verify three systems against a set of properties. We systematically compare the results with the time needed by the FTS algorithms without CEGAR to verify the same systems against the same properties. The first system is an fPromela model of the minepump system SPL (see Chapter 5 and [126, 59, 56] for more information). The SPL consists of 11 features and 128 valid products.

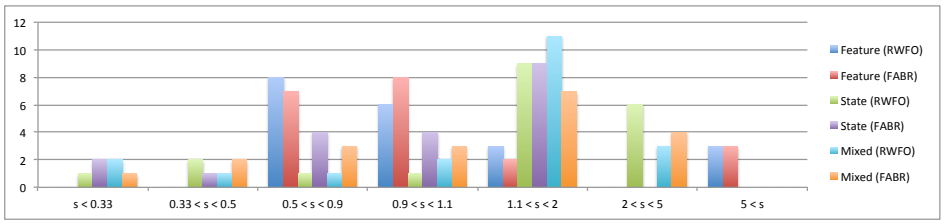


Figure 6.5: Speedups for the minepump case study.

To explore the underlying FTS, visiting 250,561 states is required. The second SPL is an elevator model inspired from Plath and Ryan [147, 58]. It is composed of eight features, which can be combined into 256 different products, and its FTS has 58,945,690 states to explore. The third and last SPL is a case study inspired by the CCSDS File Delivery Protocol (CFDP) [61], which has been re-engineered as a product line [29]. The FTS modelling the protocol consists of 1,801,581 states to explore and 56 products.

We focus first on the minepump SPL. We checked the corresponding FTS against 20 properties, including deadlock freedom, safety and liveness properties. Among these, four are satisfied by all the 128 products; one by 112 products; one by 100 products; seven by 96 products; two by 64 products; two by 56 products; and three are violated by all products. For clarity, we do not display detailed results. However, all the results together with our case studies and our implementation are available on our website.[1] All benchmarks were run on a MacBook Pro with a 2,8 GHz Intel Core i7 processor and 8 GB of DDR3 RAM. We coded an automated script to execute them. To avoid random variations, we repeated each experiment five times and computed the average. We generally discuss the results in terms of *speedup*, i.e. the verification time using a given abstraction divided by the verification time of the standard FTS algorithm. The verification time includes the time to parse the fPromela model, to build the initial FTS, and to run the verification procedure.

The results are presented in Figure 6.5, which shows the speedup of each CEGAR strategy and abstraction with respect to the standard algorithm. When feature abstraction is applied together with the RWFO strategy, the model checker performs significantly better than the standard FTS algorithm (i.e. with a speedup comprised between 1.20 and 10.67) in six cases out of 20. it has almost no effect (speedup between 0.9 and 1.1) in eight cases; and it performs slightly worse (speedup between 0.77 and 0.86) in the last six cases. The results are very similar when the FABR strategy is followed instead. In both cases, after an in-depth analysis of the returned counterexamples we noticed that the cases where the abstraction does not

[1] https://projects.info.unamur.be/fts/

improve the performance occur when every feature has to be known for the counterexamples to be triggered. This means that after several refinements the model checker ends up verifying the original concrete FTS. The small loss of performance is due to the verifications performed before the last refinement. An impressive speedup of more than nine is achieved in two of the four cases where the property is satisfied by all products; in the other two properties of this kind, the abstraction creates additional traces that lead to spurious counterexamples available to all products, which triggers a refinement of the abstract model into the original model. When only one feature is responsible for the counterexamples (i.e. the two cases where the property is satisfied by half the products), the abstraction still brings nice performance gains (speedups of 1.20 and 1.90). The cases where this abstraction technique is useless or inefficient are those where counterexamples require the knowledge of all features, independently of how many products actually violate the property. A change of strategy brings but small variations in the result; this factor thus does not seem to impact the overall efficiency of feature abstraction.

The topology of the system, however, substantially affects the efficiency of this form of abstraction. In the elevator system, each feature leads to behavioural variations that occur at the beginning of the execution of the system; yet all the products have a lot of common behaviour. This implies that late splitting occurs early in the verification and that the algorithm has to explore more than once a large part of the state space. By abstracting feature expressions, we can drastically reduce these re-explorations. Concretely, the abstraction performed significantly better when verifying six properties out of 19, achieving impressive speedups (82.12 and 93.38) in the two cases where the property is satisfied by all products. In these cases, the verification time was reduced from 293.17 and 984.32 to 3.57 and 10.54 seconds, respectively. In four cases where only one to three features are needed to discover the counterexamples, the abstraction still performs efficiently (speedup between 1.14 and 4.71). As for the remaining 13 properties, almost no variation with respect to the standard algorithm were reported. Once again, this is because all features need to be known to discover the counterexamples; this is particularly true when all products violate the property. Contrary to the elevator, the CFDP case study gives a lot of trouble to feature abstraction. The FTS modelling the protocol is uncommon, as very large parts of the state space is available to only one distinct product. Abstracting feature expression results in the addition of such a large behaviour to all the other products, which will be explored for each of them. For one property out of six, this inconvenience did not affect the results. However, it led to a significant decrease in efficiency in the five other cases, with a speedup between 0.14 and 0.46.

State abstraction appears more invasive than feature abstraction. When combined with the FABR strategy, it had no significant effect on the per-

formance for only two properties of the minepump SPL; it improved it 11 times, and worsened it seven times. In the latter cases, the loss in efficiency is substantial (speedup between 0.12 and 0.79), whereas the improvements yield speedups between 1.17 and 2.50. Our observations tend to indicate that RWFO is more appropriate for state abstraction. Indeed, the performance of the abstraction is always increased when combined with this strategy. With respect to the standard algorithm, the verification is improved for 15 properties (speedup between 1.35 and 2.57), worsened for four properties (speedup between 0.27 and 0.68), and almost unchanged for one property. Unlike feature abstraction, there seems to exist no link between the efficiency of state abstraction and the number of satisfying products: the abstraction may perform well or badly in any situation. These conclusions are confirmed by the elevator and CFDP case studies, from which we gathered similar results.

It is noteworthy that the above two abstractions have both a negative impact on verification time for only one property. Moreover, it happened several times that one form of abstraction was very inefficient but not the other. This implies that (1) there almost always exists a good choice regarding the abstraction to apply and (2) their combined effects cannot be inferred from their individual performance results. Therefore we carried out additional experiments on mixed abstraction. It turned out that the results follow the same trend as in the case of state abstraction, although mixed abstraction is slightly less efficient on average. State abstraction thus seems to cancel the effect of feature abstraction, be it negative or positive. This corroborates our previous observation that state abstraction has more impact on performance. An in-depth look at the execution times revealed that the computation of successor states in state abstraction is far more costly than the re-exploration of already computed states. Since feature abstraction improves the latter but not the former, its effects are not apparent in mixed abstraction. The other two case studies tend to confirm these conclusions. In particular, the negative effects of feature abstraction on the CFDP model are offset by the benefits of state abstraction, although mixed abstraction remains seemingly less efficient than state abstraction alone in this case.

6.4 Perspectives

All these experiments allow us to draw conclusions regarding which abstraction to use. State abstraction brings substantial benefits in most cases and should be the preferred form of abstraction. This implies that recent techniques designed for single-system abstraction are worth studying for SPLs. Also, feature abstraction can achieve dramatic reductions in verification time in particular situations, notably when the property to check is (expected to be) satisfied by all products. In the other cases, state abstraction should be

considered since it reduced the average verification time in our experiments. In the minepump case study, there was only one property where both abstractions were inefficient. Using this strategy yields an average speedup of 2.30. As for mixed abstraction, its performance is of the same order as state abstraction, but is less efficient on average. Yet, mixed abstractions offer the highest level of customization; it could thus be possible to define other mixed abstractions that are more efficient than the one we used.

Part III

Beyond Boolean Product Lines

Chapter 7

Real-Time Product Lines

Nowadays, SPLs are embedded in an increasing variety of systems, including in critical areas such as avionics and telecommunications. These systems interact with their environment in real time, and therefore have to cope with hard real-time requirements. Unfortunately, dedicated SPL model-checking techniques such as FTS, MTS, and multi-valued model checking cannot handle SPLs whose behaviour depends on real-time constraints. As an answer to these limitations, we propose an approach to model and verify real-time SPLs. Our technique relies on a combination of FTS with established real-time model-checking methods.

7.1 Timed Automata

Timed Automata (TA) [82] have become a popular formalism for reasoning over the behaviour of continuous-time systems. The main reasons are that TA model-checking remains decidable for branching-time logics such as CTL and that efficient solutions can be implemented. Moreover, renown implementations like UPPAAL [23] managed to catch the interest of academics and practitioners alike. This increases confidence that TA are suitable for verifying properties of real-time SPLs as well. Fundamentally, TA are TS that can remain in a given state only a certain amount of time, and that can execute a transition within a certain time interval. Time is represented by clocks whose value evolves continuously. Clocks can be regarded as chronometers: their value can be inspected and reset, but not modified arbitrarily. Conditions over clock values are named *clock constraints* and take the following form.

Definition 7.1 (Clock constraint)
[16] A clock constraint over a set C of clocks is formed according to the

following grammar:

$$g ::= \top \mid n \sim c \mid g \wedge g$$

with $n \in \mathbb{N}$, $c \in C$, *and* $\sim \in \{<, \leq, \geq, >\}$.

We denote by $CC(C)$ the set of clock constraints over C. In TA, a clock constraint can label a state or a transition. In the first case, the constraint is an *invariant*, which defines the interval of time in which the system can be in the state. In the second case, it is a *transition guard* specifying the interval of time during which the system can execute the transition. This leads us to the formal definition of TA.

Definition 7.2 (Timed automata)
A TA is a tuple $(Loc, Act, C, Trans, Loc_0, Inv, AP, L)$ *where*

- Loc *is a set of locations;*
- Act *is a set of actions;*
- C *is a set of clocks;*
- $Trans \subseteq Loc \times CC(C) \times Act \times 2^C \times Loc$ *is a transition relation. For a transition* (l, g, α, Res, l'), l *is the starting location,* g *is the transition guard,* α *is the action triggering the transition,* Res *is the subset of clocks to reset, and* l' *is the target location.*
- $Loc_0 \subseteq Loc$ *is a set of initial locations;*
- $Inv : Loc \to CC(C)$ *is a total function associating a location with an invariant;*
- AP *is a set of atomic propositions;*
- $L : Loc \to 2^{AP}$ *is a total function associating a location with the atomic propositions satisfied in that location.*

where $CC(C)$ *denotes the set of clock constraints over* C.

Example 7.1.1 *An example of TA is given in Figure 7.1. Similarly to TS, the possible states of the modelled system are represented by a set of* locations *(`off` and `on`). Transitions between locations, modelled by arrows, describe how the state of the system can evolve. For example, if the system*

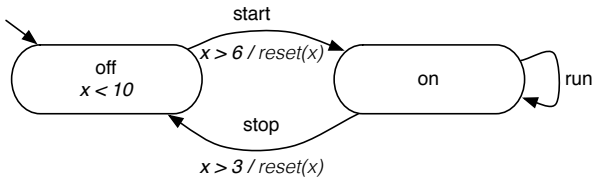


Figure 7.1: An example of Timed Automaton.

is in location `off` *and triggers the action* `start`*, it will reach location* `on`*. We use clock constraints to restrict the intervals of time during which the system can idle in a location or execute a transition to leave this location. For instance, the system can remain in location* `off` *only when the value of clock* `x` *is less than 10. Similarly, it can move to location* `on` *when the value of* `x` *is greater than 6. Overall, it means that the system always remain in location* `off` *between 6 and 10 time units. Upon the execution of a transition, the value of clocks can be reset to zero (e.g., see transitions* `start` *and* `stop`*).*

The semantics of a TA is commonly defined as an infinite TS whose states consist of a location and a valuation of the clocks. The transitions in this TS can be categorized into two types. *Delay transitions* do not change the location of the system; they only represent the passage of time. A delay transition may occur only if the invariant of the current location is still satisfied after the delay modelled by the transition. *Discrete transitions* occur when the system moves from one location to another. These are executable only when the current clock values satisfy both the guard of the executed transition and the invariant of the target location. After the execution of such transitions, clock value can be reset.

Definition 7.3 (TA Semantics)
[16] Let $ta = (Loc, Act, C, Trans, Loc_0, Inv, AP, L)$ *be a TA. The semantics of* ta *is the semantics of the TS* $(S, Act', Trans', I, AP', L')$*, noted* $TS(ta)$*, and such that*

- $S = Loc \times Val(C)$;
- $Act' = Act \cup \mathbb{R}_{\geq 0}$;
- $I = Loc_0 \times \eta_0$;
- $AP' = AP \cup CC(C)$;
- $L'(l, \eta) = L(l) \cup \{cc \in CC(C) \mid \eta \models cc\}$.

where $Val(C)$ the set of clock evaluations, that is, the set of total functions $\eta : C \to \mathbb{R}^+$ that assign a non-negative real value to every clock, and η_0 is such that $\forall c \in C \bullet \eta_0(c) = 0$.

Temporal logics such as LTL and CTL cannot express real-time properties. Instead, one can specify them in Timed Computation Tree Logic [7] (TCTL), a timed variant of CTL.

Definition 7.4 (TCTL)
[7, 16] A TCTL formula Φ over a set AP of atomic propositions and a set C of clocks is formed according to the following grammar:

$$\Phi ::= \top \mid a \mid cc \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi_1 \mid \exists(\Phi_1 \mathrm{U}_J \Phi_2) \mid \forall(\Phi_1 \mathrm{U}_J \Phi_2)$$

where Φ_1 and Φ_2 are TCTL formulae, $a \in AP$, $cc \in CC(C)$, and J is a subinterval of $[0, \infty)$. When $J = [0, \infty)$, it can be omitted.

In this work, we consider a specific fragment of TCTL that can be checked through reachability analysis, as we deal with this type of analysis in the subsequent sections. Our approach is, however, not limited to this fragment and can straightforwardly be extended to fully support TCTL. A formula of this fragment has the form

$$\Phi ::= \exists\Diamond_J \phi \mid \exists\Box_J \phi \mid \forall\Diamond_J \phi \mid \forall\Box_J \phi \mid \forall\Box(\phi \to \exists\Diamond_J \psi)$$

where ϕ and ψ are Boolean formulae over the set of atomic propositions, $\Diamond_J \phi = \top \mathrm{U}_J \phi$, and $\forall\Box_J \varphi = \neg\exists_J \neg\varphi$. Intuitively, the formula $\exists\Diamond_J \phi$ is satisfied if and only if *"there exists a path where a state satisfying ϕ is reached, and this state can be reached in a time that lies in J"*. Similarly, the formula $\exists\Box_J \phi$ expresses the requirement that *"there exists a path where, during the interval of time J, the proposition ϕ is always satisfied"*. When the symbol $\forall$ replaces $\exists$, it means that the property that follows must hold for every path. Finally, $\forall\Box(\phi \to \exists\Diamond_J \psi)$ means that *"whenever ϕ occurs, there must be a way to satisfy ψ within the interval of time J"*. This last formula is notably helpful to express real-time fault recovery requirements.

Model checking a TA against a TCTL formula is achieved by computing the reachability relation in the underlying infinite TS. Should that model not be infinite, this would come down to model-checking a standard TS. The infinite size comes from the definition of the state space itself, that is, the cartesian product of the set of locations with the set of clock valuations.

The former is finite but the latter is not. This is the reason why all the existing TA model-checking algorithms make use of *time abstraction*. One of the most popular mechanism for abstracting time consists in representing a set of clock valuations with a *clock zone* [82]. Given a clock constraint g, a clock zone for g is the maximal set of clock valuations satisfying g. In practice, zones can be represented as the conjunction of constraints of the form $x - y < n$ with $n \in \mathbb{Z}$ and $x, y \in C \cup \{0\}$. Several TA model-checkers, notably UPPAAL [23], rely on this representation. Applying zone-based abstraction on the infinite TS yields a finite TS where states are couples of location and zone. One can thus compute the set of reachable states in this TS and consequently determine if and when the system can reach a given location [82]. We apply the principles of this exploration procedure in our own verification algorithms for real-time SPLs, which are presented further in this chapter.

7.2 Modelling Real-Time SPL Behaviour

In real-time SPLs, features may influence the behaviour of a system in two ways. First, features may add or remove capabilities like in discrete SPLs. Second, they may also influence the time constraints that determine the minimum and maximum delays for executing a given action or remaining in a given state. Accordingly, we extend TA with variability in the same way as TS were extended to FTS. The resulting formalism is named Featured Timed Automaton (FTA). Before giving its formal definition, we first present it through an introductory example.

Example 7.2.1 *We consider a simplification of the well-known mine pump example [126] extended with real-time. For now, we focus on a closed model of the software controlling the pump itself and ignore the other components of the system. We assume that the software is developed as an SPL with three optional features: Button (B), FastStart (Fstart), and FastStop (Fstop). Button defines that the pump is equipped with a button used to start and stop the engine. If FastStart (resp. FastStop) is enabled, then less time is required for the pump to start (resp. stop). This small SPL has a total of eight products.*

Figure 7.2 shows a graphical representation of an FTA modelling the real-time behaviour of the pump SPL. As in TA, we model the locations of the system. Here, there are only two locations, `on` *and* `off`*, which models that the pump is started and stopped, respectively. The pump can transit between these locations if and only if it has feature B. Initially, the pump is in the* `off` *location. At some point, the pump must be started, that is, the pump cannot remain in the* `off` *location indefinitely. As for TA, we model this restriction with a location invariant. In FTA, however, invariants may*

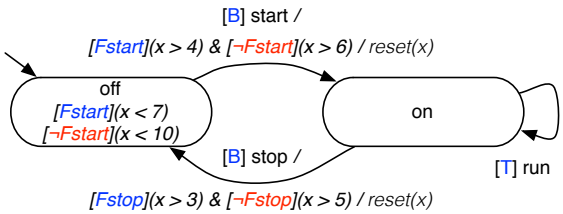


Figure 7.2: FTA modelling the pump SPL.

depend on variability. For example, we should be able to express the following requirement :

> *"The maximum amount of time the pump may remain in* `off` *location depends on the presence of FastStart is enabled. With this feature, the maximum time is 7 time units, otherwise it is 10 time units."*

In order to model the above requirement, we combine features with location invariants. More precisely, we define an invariant as the conjunction of clock constraints annotated with a feature expression. We name this construct featured clock constraint*. Then, only the products satisfying the feature expression have to satisfy the associated clock constraint. The feature expression thus plays the same role as the feature quantifier in fLTL (see Definition 3.5). In our example, the invariant of location* `off` *is given by* $x < 7$ *for the products having the feature FastStart; for the others, the invariant is given by* $x < 10$*. Accordingly, we specify the invariant of location* `off` *as*

$$[FStart](x < 7) \wedge [\neg FStart](x < 10).$$

As for invariants, variability influences the executability of transitions and their associated clock constraints. As mentioned above, the feature Button is needed to execute the transitions `start` *and* `stop`*. Consequently, these transitions are annotated with the feature expression* B*. On the contrary, the transition* `run` *does not require the feature Button to be executed and is thus annotated with the feature expression* $\top$*, which is satisfied by every product. Another requirement for the pump is the need for a preheating time before it begins to run. This delay depends on feature FastStart. We can express that constraint as follows :*

> *"The pump must remain at least in 6 time units in the* `off` *location before starting. If the pump is equipped with feature FastStart, this delay is reduced to 4 time units."*

As before, we may model this requirement by guarding the transition with a featured clock constraint. In this case, the constraint is

$$[FStart](x > 4) \wedge [\neg FStart](x > 6).$$

The transition `stop` *is constrained similarly. We thus annotate that transition with another featured clock constraint. Overall this model defines that the pump always remains in location* `off` *between 4 and 7 time units if feature FastStart is enabled, and between 6 and 10 time units otherwise.*

7.2.1 Syntax and Semantics of FTA

As our introductory example suggests, the encoding of variability within FTA relies on a combination of feature expressions and clock constraints. The resulting formalisms is defined as follows.

Definition 7.5 (Featured clock constraint)
A featured clock constraint over a set F of features and a set C of clocks is a formula of the form

$$g ::= \top \mid [\chi](c \sim n) \mid g \wedge g$$

where $c \in C, n \in \mathbb{N}$, $\sim \in \{<, \leq, \geq, >\}$, and χ is a feature expression over F.

Intuitively, $g = [\chi]g'$ means that g' must hold for products satisfying χ – the others do not have to satisfy the constraint. We denote by $FCC(F, C)$ the set of featured clock constraints over F and C. The satisfiability relation for featured clock constraints by a couple $(p, \eta) \in 2^F \times Val(C)$ is recursively defined as follows:

$$(p, \eta) \models g \Leftrightarrow \begin{cases} (p, \eta) \models g_1 \wedge (p, \eta) \models g_2, & g = g_1 \wedge g_2, \\ p \not\models \chi \vee \eta \models g', & g = [\chi]g'. \end{cases}$$

When $p \not\models \chi$, p is said to trivially satisfy any featured clock constraint of the form $[\chi]g$. Note that a classical (non-featured) clock constraint g can be regarded as a featured clock constraint where each feature expression is a tautology. Also, we allow the use of $[\chi](g_1 \wedge g_2)$ as a shortcut for $[\chi]g_1 \wedge [\chi]g_2$. As an example, the featured clock constraint $[FastStart](x > 4)$ (see Figure 7.2) is satisfied by a product p and a clock valuation η if and only if p does not have feature $FastStart$ or clock `x` has a value higher than 4 in η.

Feature expressions and featured clock constraints allow us to model the impact of features on real-time models. By combining them with timed

automata, we obtain a new formalism to model the behaviour of real-time SPLs.

Definition 7.6 (Featured timed automaton (syntax))
A featured timed automaton is a tuple $(Loc,\ Act,\ C, trans,\ Loc_0, Inv,\ AP, L,\ fm, \gamma)$ *such that*

- Loc *is a finite set of locations,*
- Act *a finite set of actions,*
- C *is a finite set of clocks,*
- $Loc_0 \in Loc$*, is the set of initial locations,*
- $Trans \subseteq Loc \times FCC(F, C) \times Act \times 2^C \times Loc$ *is a finite set of transitions,*
- $Inv : Loc \to FCC(F, C)$ *is a total invariant function that associates featured clock constraints to locations,*
- AP *is a set of atomic propositions,*
- $L : Loc \to AP$ *is a total function labelling locations with atomic propositions,*
- fm *is a feature model over a set* F *of features,*
- $\gamma : Trans \to 2^{(2^F)}$ *is a total function that labels transitions with feature expressions.*

According to this definition, we consider that a feature may modify the availability of transitions, transition guards, and location invariants. As for FTS, the function γ associates a transition t with a feature expression such that $\gamma(t)$ encodes the set of products able to execute t. We model the variability of transition guards and location invariants with featured clock constraints, as opposed to classical clock constraints. For example, if $[\neg FastStart](x < 10)$ is a featured clock constraint occurring in the invariant of a location, then the system is forbidden to be in that location when the value of x is greater than 10. This prohibition, however, holds *only* for the products that do *not* have the feature $FastStart$. One can easily relate the above definition with the example shown in Figure 7.2. However, be aware that we have omit to display the atomic propositions in order to increase the readability of the figure.

Thanks to the above constructs, FTA is a convenient formalism to model

the real-time behaviour of a set of products. Besides, from an FTA we can derive a TA that models the behaviour of a specific product. To achieve this, we define a *projection* operator over FTA. Basically, the projection of an FTA onto a product p is obtained by (1) removing the transitions unavailable to p, (2) replacing any featured clock constraint $[\chi]g$ by g if $\chi(p) = \top$ holds, or by $\top$ otherwise.

Definition 7.7 (Projection of FTA)
The projection of an FTA $fta = (Loc, Act, C, Trans, Loc_0, Inv, AP, L, d, \gamma)$ *onto a valid product* $p \in [\![fm]\!]$ *is the TA* $fta_{|p} = (Loc, Act, C, Trans', Loc_0, Inv', AP, L)$ *with*

$$\forall l \in Loc \bullet Inv'(l) = Inv(l)_{|p}$$
$$Trans' = \{t = (l, g_{|p}, \alpha, R, l') \mid t \in Trans \wedge p \models \gamma(t)\}$$

where the projection of a featured clock constraint g *onto a product* p *is recursively defined as*

$$g_{|p} = \begin{cases} (g_1)_{|p} \wedge (g_2)_{|p}, & g = g_1 \wedge g_2, \\ g', & (g = [\chi]g') \wedge p \models \chi, \\ \top & (g = [\chi]g') \wedge p \not\models \chi. \end{cases}$$

For example, Figure 7.1 is the TA resulting from the projection of the FTA shown in Figure 7.2 to the product $\{B, Fstop\}$.

Each TA resulting from the projection of an FTA onto a specific product exactly models the behaviour of that product. Given that an FTA models a set products, we define its semantics as a function that associates every valid product with the semantics of its projection.

Definition 7.8 (Featured timed automaton (semantics))
The semantics of an FTA $fta = (Loc, Act, C, trans, Loc_0, Inv, AP, L, d, \gamma)$ *is the function* $[\![fta]\!]$ *such that*

$$\forall p \in [\![fm]\!] \bullet [\![fta]\!](p) = [\![TS(fta_{|p})]\!].$$

For instance, the FTA presented in Figure 7.2 associates the product $\{B, Fstop\}$ to the TA shown in Figure 7.1.

7.3 An Alternative Definition of FTA

Definition 7.8 expresses the semantics of FTA in terms of TA, the semantics of the latter being itself defined as an infinite induced TS. In the end, the above semantics can be seen as a function from FTA to TS where variability is first resolved (through the definition of projection) before real-time is unfolded. Alternatively, one could invert these two processes and first transform an FTA into an infinite FTS, whose semantics is defined as the projection onto a set of products. In this section, we investigate this alternative semantics and prove that it is equivalent to the previous one. Before giving the actual definition, it is first required to perform some transformations on FTA. We formalise these transformations and show that they preserve FTA semantics.

7.3.1 Separating Variability from Real-Time

Consider again the FTA shown in Figure 7.2. We observe that the transition *start* from off to *on* could have been defined without the use of featured clock constraints. In this case, the transition is equivalent to two alternative transitions, i.e. one for the products having the feature $FastStart$ with the guard $x > 4$, and one for the other products with the guard $x > 6$ (see Figure 7.3 for an illustration). The above reasoning thus raises the question of studying a possibly equivalent form of FTA where feature expressions have been removed from featured clock constraints. In what follows, we show that we can transform every FTA into an equivalent FTA without featured clock constraints. Depending on whether we consider transition guards or location invariants, the transformation procedure is different.

Let us first consider the first case (e.g. the transition `start` in Figure 7.2). Let $t = (l, g, \alpha, Res, l')$ be a transition. The idea of the transformation is to create copies of the transition such that a given product p can execute only one of the copies and the guard of this copy is a classical clock constraint equivalent to the projection of the guard of the original transition onto p. The original transition is then removed. The transformation is achieved as follows:

1. Separate products into groups such that two products p_1 and p_2 are in the same groups if and only if the projections of the transition guard onto p_1 and p_2 are equivalent. This results in a set of set of products $\{G_1, \ldots, G_k\} \subseteq 2^{(2^F)}$ such that G_i is the largest set satisfying

$$\frac{c_{|p} = c_i}{p \in G_i}$$

   where c_i is the clock constraint resulting from the projection of the guard onto any product in G_i.

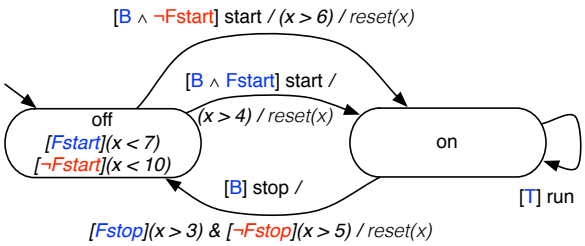


Figure 7.3: FTA with a transformed transition guard.

2. Remove t from $Trans$.

3. For each group G_i add $t_i = (l, c_i, \alpha, Res, l')$ in $Trans$ and define $\gamma(t_i) = \gamma(t) \wedge \mathbb{B}(G_i) \wedge \bigwedge_{j \neq i} \neg\mathbb{B}(G_j)$.

For the transition `start`, we have two groups, namely the products that have feature $Fstart$ and the ones that do not. The corresponding clock constraints are $x > 4$ and $x > 6$, respectively. As shown in Figure 7.3, this results in two new transitions: the first has the guard $x > 4$ and is executable by the products satisfying $B \wedge Fstart$; the second has the guard $x > 6$ and is executable by the products satisfying $B \wedge \neg Fstart$. This new FTA has the same semantics as the one shown in Figure 7.2. This transformation preserves the semantics of the model by definition of the grouping criteria and of the projection for transition guards.

The second transformation, which removes features from a location invariant, also relies on duplication. However, we now have to duplicate both the location and its incoming transitions. Let l be this location. Like we did in the previous transformation, we separate the products into groups $G_1, \ldots, G_k$ with the corresponding projections $c_1, \ldots, c_k$ of the location invariant. Then we perform the following steps:

1. If l is in Loc_0, add a new location l_0 in Loc_0 with

$$Inv(l_0) = \bigwedge_{c \in C} c \leq 0.$$

2. For each group G_i do:

   (a) Create a copy l_i of l, add it in Loc and remove l.

   (b) For every $t = (l, g, \alpha, Res, l') \in Trans$, remove it and add $t_i = (l_i, g, \alpha, Res, l')$. Set $Inv(l_i) = c_i$ and $\gamma(t_i) = \gamma(t)$.

   (c) For every $t = (l', g, \alpha, Res, l) \in Trans$, remove it and add $t_i = (l', g, \alpha, Res, l_i)$. Define $\gamma(t_i) = \gamma(t) \wedge \mathbb{B}(G_i)$.

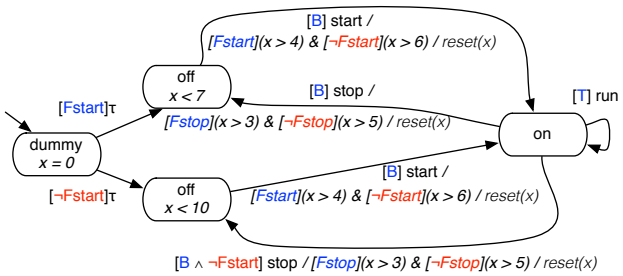


Figure 7.4: FTA with a transformed invariant.

(d) If l was in Loc_0, add τ in Act and $t_i = (l_0, \top, \tau, \emptyset, l_i)$ in $trans$, and set $\gamma(t_i) = \mathbb{B}(G_i)$.

The result of applying this transformation to location off is illustrated in Figure 7.4. In this new model, we have two copies of the location: the upper one corresponds to products having the feature $FastStart$; the lower one corresponds to the other products. Since off was an initial location, a dummy location l_0 has been added. This is needed to ensure that any product starts in a location with an appropriate invariant. Provided that no semantics is associated to the dummy location and its outgoing transitions, this FTA is equivalent to the one shown in Figure 7.2 modulo weak timed bisimulation of the respective projections [164]. Note that the above two transformations are commutative; their successful application yields the FTA shown in Figure 7.5.

From the above observations, we conclude that the sole purpose of featured clock constraints is to increase the conciseness of the model through a reduction in the number of transitions and locations. More precisely, both transformations multiply the number of transitions by a factor of $\mathcal{O}(2^{|F|})$, even though they decrease the size of the guards and invariants by the same factor. Additionally, The second transformation yields models where the number of locations is also multiplied by the same factor. Although it considers but a small example, Figure 7.5 already gives account for that phenomenon.

7.3.2 An Infinite FTS Semantics

We can finally provide FTA with a new semantics, namely in terms of FTS. This alternate semantics is a generalization of the transformation from TA to TS [16] combined with the aforementioned two FTA transformations. More precisely, we define a transformation of FTA into an *induced* FTS. The initial step consists in removing all featured clock constraints as presented above. It results an FTA with classical clock constraints from which the induced

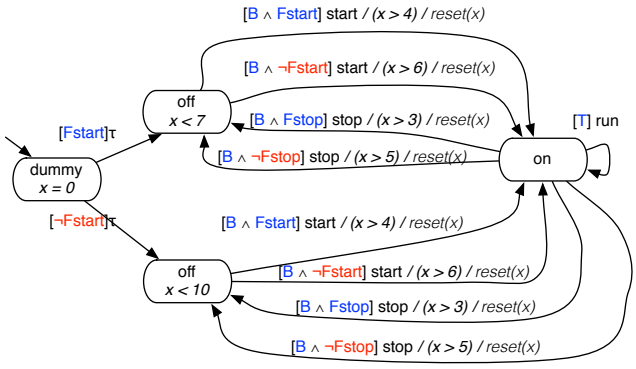


Figure 7.5: FTA with transformed invariants and guards.

FTS is extracted just like an infinite-state TS is produced from a TA.

First, a state of an induced FTS must refer to a precise point of time. Consequently, the whole state space is defined as $Loc \times Val(C)$ and is infinite. An initial state is then an initial location with every clock value set to 0:

$$I = Loc_0 \times \{\eta \in Val(C) \mid \forall c \in C \bullet \eta(c) = 0\}.$$

To be able to relate a state with the time constraints it satisfies, we augment the set of atomic propositions with the set of clock constraints occurring in the original FTA, which we denote by $AP' = AP \cup CC(C, FTA)$. We then update the function that associates each state with atomic propositions according to this definition of AP'. More formally, this new function is defined as $L' : Loc \times Val(C) \to 2^{AP'}$:

$$L'(l, \eta) = L(l) \cup \{cc \in CC(C, FTA) \mid \eta \models cc\}.$$

Finally, the set of transitions is composed of discrete and delay transitions. Discrete transitions are the result of a modification in the system location, namely the execution of an actual transition in the original FTA. The state reached by a discrete transition is then determined according to:

1. the clock valuations of the state from which the transition is executed;
2. the guard of the corresponding transition in the FTA;
3. the set of clocks to reset;
4. the invariant of the target location.

Formally, $t' = ((l, \eta), \alpha, (l', \eta'))$ is a discrete transition if and only if

$$\exists t \in trans \bullet t = (l, g, \alpha, Res, l') \ \wedge \ \eta \models g \\ \wedge \ \eta' = \text{reset } Res \text{ in } \eta \ \wedge \ \eta' \models Inv(l')$$

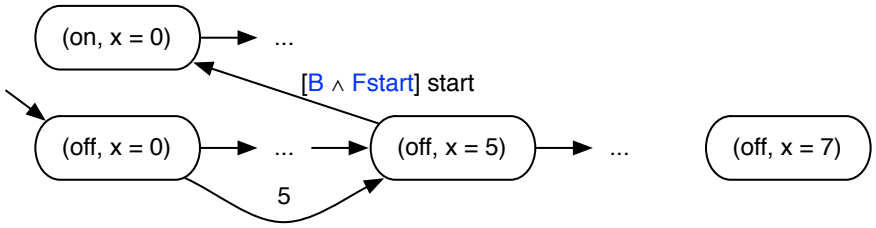


Figure 7.6: An infinite induced FTS.

where reset Res in η is a clock valuation function η' such that

$$\forall c \in C \bullet \eta'(c) = \begin{cases} 0, & c \in Res, \\ \eta(c), & \text{otherwise.} \end{cases}$$

The feature expression labelling a discrete transition is equal to the one labelling the corresponding transition in the FTA, that is, $\gamma'(t') = \gamma(t)$. Delay transitions models the passage of time when the system remains in a given location. Intuitively, a delay transition may occur if the duration of the delay does not lead to a violation of the current location invariant. Thus, $t' = ((l, \eta), d, (l, \eta + d))$ is a delay transition if and only if $\eta + d \models Inv(l)$, where $\eta + d$ is η with the value of every clock increased by d time units. Such a transition is executable by all the products: $\gamma(t') = \top$.

Example 7.3.1 *We partially illustrate the result of this translation in Figure 7.6, where it has been applied on the FTA shown in Figure 7.2. The initial state is composed of the location* `off` *and the clock valuation where* x *has the value 0. This state has an uncountable number of outgoing delay transitions (that is, to every state such that* $x < 7$*). When the value of* x *is greater than 4, a transition to the state* `(on, x = 0)` *is executable for the products that have the features Button and FastStart.*

So far, we have equipped FTA with two different semantics. Given that a TA is nothing more than an infinite TS and that an FTS is a function from the set of products to TS [64], both end up describing FTA as a function that associates a product with the behaviour of an infinite TS. These two semantics are equivalent, i.e. for every valid product p, they give rise to TS with equivalent behaviour.

Theorem 7.1
Let fta be an FTA defined over an FM fm and $FTS(fta)$ its induced infinite FTS. Let $TS(ta)$ denote the infinite TS induced by a TA ta.

Then,

$$\forall p \in [\![fm]\!] \bullet [\![FTS(fta)_{|p}]\!] = [\![TS(fta_{|p})]\!].$$

PROOF. *Immediately follows from Definition 3.3, Definition 7.7, and the rules of the transformation from FTA to FTS.* □

These different formulations of FTA semantics reflects the existence of two distinct approaches for FTA model checking.

7.4 Algorithms

We have presented the FTA formalism, its syntax and two semantics, which end up being equivalent. The purpose of providing multiple definitions lies in the existence of two distinct methods for model-checking a real-time SPL modelled as an FTA. For a formula Φ and an FTA fta defined over an FM fm, we define the F-satisfiability of fta with respect to ϕ as the feature expression noted $fta \models \phi$ and defined such that

$$(fta \models \phi)(p) = (fta_{|p} \models \phi).$$

A first, product-based method to compute this consists in computing the projection of the FTA onto each valid product, thereby producing a set of TA, and checking all these TA against the property. An important weakness of this verification approach is its inability to take into account that multiple products can share common behaviour. To overcome this limitation, we propose another method that combines the abstraction of time using zones with the efficient algorithms previously designed for FTS. We illustrate the use of this approach to compute reachability relations. The notion of reachability in FTA is different than that of TA and FTS, because it has to consider (1) the reachable locations, (2) the products able to reach them, and (3) the values of the clocks when these locations are reached.

To compute reachability, we first define a new successor operator which takes both variability and time constraints into account. Since features do not occur in invariants and transition guards in the separated form (see Section 7.3.1), the clock zone that is reached upon the execution of a transition does not depend on the features. Therefore, this successor zone is determined according to the current zone ζ, the current location l, and the chosen transition $t = (l, g, \alpha, R, l')$. It is given by

$$next(l, \zeta, t) = \text{reset } R \text{ in } \left((\zeta \wedge Inv(l))^{\Uparrow} \wedge Inv(l) \wedge g\right)$$

where reset R in ζ' is the zone ζ' where every clock in R has been reset to 0 and $(\zeta')^{\Uparrow}$ is the zone ζ' after any elapsing of time [82]. The zone $next(l, \zeta, t)$

can be empty if $\zeta^{\Uparrow}$, the invariant of l and the guard of t are incompatible. In this case, we must discard it and not pursue the exploration. Similarly, since the set of features of a given product remains constant over time, the satisfiability of a feature expression annotating a transition does not depend on the value of clocks. These two properties show that Boolean features and real-time constraints are completely orthogonal constructs. *Ipso facto*, defining the successor operator comes down to determining, given a transition, which products are able to execute it and which zone is reached after the execution.

Definition 7.9 (Successor relation in FTA)
Let fta Let $fta = (Loc, Act, C, Trans, Loc_0, Inv, AP, L, fm, \gamma)$ be an FTA, fts its induced FTS. The successor relation of fts is a relation $Post \subseteq (Loc \times 2^{Val(C)}) \times (Loc \times 2^{Val(C)}) \times 2^{(2^F)}$ defined as

$$((l,\zeta),(l',\zeta'),\gamma) \in Post \Leftrightarrow \forall p \in [\![\gamma]\!] \\ \exists t \bullet t = (l,g,\alpha,R,l') \in Trans \bullet \gamma(t)(p) \wedge \zeta' = next(\zeta,l,t).$$

The successor relation as defined above is not necessarily minimal with respect to the inclusion of zones and sets of products. This is an issue we have to deal with during the exploration of the state space of the induced FTS. We next derive the definition of reachability in FTS from the successor relation.

Definition 7.10 (Reachability relation in FTA)
Let fta be an FTA. The reachability relation in fta is a set of triplets $R(fta) \subseteq Loc \times 2^{Val(C)} \times 2^{(2^F)}$ such that $(l_k,\zeta_k,b_k) \in R(fta) \Leftrightarrow \exists (l_0,\zeta_0,b_0)\dots(l_k,\zeta_k,b_k)$ where $l_0 \in Loc_0$, $\zeta_0 = \{\eta_0\}$, $b_0 = \top$, and for all $i < k$ and $p \in [\![fm]\!]$ we have

$$b_{i+1}(p) \Leftrightarrow b_i(p) \wedge \exists (l_{i+1},\zeta_{i+1},\gamma) \in Post(l_i,\zeta_i) \bullet \gamma(p).$$

We can thus compute the reachability relation through successive computations of the successor relation – see Algorithm 4. The initial locations are always reachable by all the valid products and for a clock zone ζ_0 where every clock has its value set to zero (Line 4). The procedure then explores the state space of the induced FTS with a depth-first search. At each iteration, it computes the successor sets of the current state (Line 9). For every successor $s = (l',\zeta',b')$, the procedure adds s in the relation if and

only if there exist no element of the relation (l', ζ'', b'') such that the products represented by b' are also represented by b'' and the clock zone ζ' is a subset of ζ''. Otherwise, it means that we already know that the products satisfying b' can reach l' in the clock zone ζ'. In this case, it is useless to consider s and to pursue the search from this state. For a similar reason, when the procedure adds a triplet in the relation it removes every triplet that has become redundant because of the addition of s (Line 11). The redundancies originate from the fact that, like the successor relation, the reachability relation is not always minimal. To overcome this, we can use a particular union operator we discuss further in this section.

Input: $fta = (Loc, Act, C, trans, Loc_0, Inv, AP, L, fm, \gamma)$.
Output: $R(fta)$.

```
1  R ← ∅;
2  Stack ← [];
3  foreach s_0 ∈ {(l_0, ζ_0, fm) | l_0 ∈ Loc_0} do
4  |   R ← R ∪ {s_0};
5  |   push(s_0, Stack);
6  end
7  while Stack ≠ [] do
8  |   (l, ζ, b) ← pop(Stack);
9  |   foreach (l', ζ', b') ∈ Post(l, ζ, b) do
10 |   |   if ∄(l', ζ'', b'') ∈ R • ⟦b'⟧ ⊆ ⟦b''⟧ ∧ ζ' ⊆ ζ'' then
11 |   |   |   R ← R ⊔ {(l', ζ', b')};
12 |   |   |   push((l', b', ζ'), Stack);
13 |   |   end
14 |   end
15 end
16 return R
```

Algorithm 4: Reachables(fta)

As for TA, a forward reachability algorithm using zone-based abstraction does not always terminate without an extrapolation operator [24]. Since termination is a classical problem already solved in TA model-checking and is independent of the addition of variability, we do not deal with it.

To increase the efficiency of this algorithm, we have to deal with a doubly-partial order, though. This issue is the inheritance of the model-checking algorithms for TA and FTS. For instance, let us suppose that the system already reached the state (l, ζ) for the products satisfying b and that it reaches a state (l, ζ') for the same products in the future. If ζ' is a subset of ζ, it is useless to continue the search from the latter state because the set of states reachable from (l, ζ') would also be a subset of the states reachable from (l, ζ). Similarly, if the system first reaches (l, ζ) for products satisfying

b, and then reaches this state for products satisfying b' with $b' \Rightarrow b$, we should not pursue the exploration: since any products satisfying b' also satisfies b, any information obtained by further exploration would be redundant. This doubly-partial order may lead to trickier phenomena, where both the zones and the feature expressions have to be compared. To cope with such cases, we define formal rules that determine with which zone and feature expression we must resume the exploration. Applying these rules reduces the number of explored states during a verification. More precisely, they guarantee that the reachability relation is an *antichain*.

Definition 7.11 (Antichain in FTA reachability relation)
Let $R \subseteq Loc \times 2^{Val(C)} \times 2^{(2^F)}$. R is an antichain if and only if

$$\forall r, r' \in R \bullet r \leq r' \Rightarrow r = r'$$

where

$$(l, \zeta, b) \leq (l', \zeta', b') \Leftrightarrow (l = l') \wedge (\zeta \subseteq \zeta') \wedge (\llbracket b \rrbracket \subseteq \llbracket b' \rrbracket).$$

To preserve the antichain upon union, a specific operator is needed. Algorithm 5 shows the procedure to add a given triplet (l, ζ, b) to an antichain R. The idea is to iterate over the elements in R to remove any redundancy contained in (l, ζ, b) with respect to the relation. Let (l, ζ', b') be an element of R. If $\zeta \subset \zeta'$ then the triplet is redundant for any product satisfying both b and b'; indeed, we already know at what time these product can reach location l. To remove this redundancy, we transform b into $b \wedge \neg b'$ (see Lines 6–9). If $\zeta \supset \zeta'$ then the redundancy lies in R. In this case, we change b' into $b' \wedge \neg b$ and we remove the element from R if b' does not encode any valid product (Lines 10 – 15). If $\zeta = \zeta'$, we retain the products in $\llbracket b' \rrbracket$ (Lines 16–18) and at the end of the algorithm, add to this set the products satisfying b that are not redundant (Line 28). Finally, any element (l', ζ', b') with $l \neq l'$ is added to R' (Lines 19–21).

In the end, our algorithm combines existing model-checking approaches. The only new construct we introduced through the whole chapter are featured clock constraints, which we could split up in feature expressions and classical clock constraints. This is reflected in the computation of the successor relation, where zones and products are considered orthogonally. These nice characteristics make an implementation considerably easier.

Input: $R \subseteq \{l\} \times 2^{Val(C)} \times 2^{(2^F)}$, $(l, \zeta, b) \in Loc \times 2^{Val(C)} \times 2^{(2^F)}$.
Output: R', an antichain such that

$$\forall r = (l, \{\eta\}, \mathbb{B}(\{p\}) \bullet \\ (\exists r' \in R' \bullet r \leq r') \Leftrightarrow (\exists r'' \in R \cup \{l, \zeta, b\} \bullet r \leq r'').$$

```
1  R' ← ∅;
2  equal ←⊥;
3  while (b ∧ fm ⊭⊥) ∧ (R ≠ ∅) do
4  |  Take (l', ζ', b') from R;
5  |  if l = l' then
6  |  |  if ζ ⊂ ζ' then
7  |  |  |  b ← b ∧ ¬b';
8  |  |  |  R' ← R' ∪ {(l, ζ', b')};
9  |  |  end
10 |  |  else if ζ ⊃ ζ' then
11 |  |  |  b' ← b' ∧ ¬b;
12 |  |  |  if b' ∧ fm ⊭⊥ then
13 |  |  |  |  R' ← R' ∪ {(l, ζ', b')};
14 |  |  |  end
15 |  |  end
16 |  |  else if ζ = ζ' then
17 |  |  |  equal ← b';
18 |  |  end
19 |  |  else
20 |  |  |  R' ← R' ∪ {(l, ζ', b')};
21 |  |  end
22 |  end
23 |  else
24 |  |  R' ← R' ∪ {(l', ζ', b')};
25 |  end
26 end
27 if equal ≠⊥ then
28 |  R' ← R' ∪ {(l, ζ, equal ∨ b)};
29 end
30 if R ≠ ∅ then
31 |  R' ← R' ∪ R;
32 end
33 return R'
```

Algorithm 5: $R \sqcup \{(l, \zeta, b)\}$

7.5 Evaluation

We implemented the algorithms presented in this chapter in ProVeLines. Our tool allows one to specify FTA in a real-time extension of fPromela (see Chapter 9), and to verify properties expressed in the aforementioned fragment of TCTL via reachability computation. From this computation, it extracts the set of products that violate the property. During the verification, the tool maintains the set of products already known to violate the checked formula. These products are then excluded from the subsequent searches. This optimisation grants a reduction in verification time as it allows the checker to avoid exploring paths that only bad products can execute. In particular, the exploration is immediately stopped when all the valid products are known to violate the property. Such optimisations have shown to bring substantial performance gains in the FTS algorithms [56].

As we have seen in the previous section, elements of the reachability relation are triplets composed of a location, a feature expression and a clock zone. In the actual implementations, we encode a state as a data structure with three fields, i.e. one for each member of the triplet. The most important part of this data structure is the zone representation. In our implementation, we represent zones by its most popular dedicated data structure, i.e., *Difference Bound Matrix* (DBM) [82]. This allows us to reuse their efficient library implementation in the UPPAAL model-checker [23]. As for feature expressions, we encode them as BDDs [32] and we use their implementation in the CUDD package.[1] Finally, we encoded locations explicitly.

Previous work on FTS have shown that variability-aware procedures are generally more efficient than product-by-product computations [59, 58, 63]. It means that the avoidance of redundant verification offsets the overhead of managing variability. If we add another dimension, namely real-time, we can expect the overhead generated by variability to become an even lower part of the total verification effort. We provide preliminary experimental results regarding the relative performance of the product-based algorithm and ours. We used ProVeLines to compute the complete reachability relation and to verify the system against 11 TCTL formulae (see Table 7.1). These formulae were declined in different variants such that two variants differ only from their time constraints. First, we measure the time needed by the tool to perform the analysis using the variability-aware algorithm. Second, given the definition of the FTA in fPromela, we used a separate script to produce each projection of the FTA into a valid product on which classical timed model checking techniques are applied. We do not measure the runtime of this script as we want to exclude the time for computing the projection from the total verification time.

For our experiments, we consider a real-time extension of the mine-pump

[1] http://visi.colorado.edu/~fabio/CUDD

Table 7.1: Properties used for the evaluation.

#1	Reachability
#2	$\forall\Box(methane \Rightarrow \exists\Diamond_{\leq 4}\ pumpOff)$
#3	$\forall\Box(methane \Rightarrow \exists\Diamond_{\leq 2}\ pumpOff)$
#4	$\forall\Box(pumpOn \Rightarrow \exists\Diamond_{\leq 15}\ lowWater)$
#5	$\forall\Box(pumpOn \Rightarrow \exists\Diamond_{\leq 10}\ lowWater)$
#6	$\forall\Box(pumpOn \Rightarrow \exists\Diamond_{\leq 5}\ lowWater)$
#7	$\forall\Box(highWater \wedge \neg methane \Rightarrow \exists\Diamond_{\leq 10}\ pumpOn)$
#8	$\forall\Box(highWater \wedge \neg methane \Rightarrow \exists\Diamond_{\leq 5}\ pumpOn)$
#9	$\forall\Box(highWater \Rightarrow \exists\Diamond_{\leq 20}\ lowWater)$
#10	$\forall\Box(highWater \Rightarrow \exists\Diamond_{\leq 15}\ lowWater)$
#11	$\forall\Box(highWater \Rightarrow \exists\Diamond_{\leq 10}\ lowWater)$
#12	$\forall\Box(highWater \Rightarrow \exists\Diamond_{\leq 5}\ lowWater)$

controller exemplar [126], of which the toy example of the previous sections is also inspired (see also Chapter 5). We declined the SPL into three versions with a different degree of refinement in the feature model. The first has six features and 12 products, the second has 10 features and 72 products, and the third has 13 features and 512 products. In this SPL, both the capabilities of the controller and the efficiency of the pump are subject to variations between products. For instance, the feature $MethaneAlarm$ determines whether or not the system is equipped with an alarm that triggers once there is methane; the feature $FastPump$ increases the running speed of the pump.

All benchmarks were run on a MacBook Pro with a 2.4 GHz Core 2 Duo processor and 4 Gb of RAM running Mac OS 10.6. To avoid the influence of other running processes, we repeated each experiment five times and computed the average. The results are shown in Table 7.2. For every variant of the SPL and every formula, we give the time needed by the product-based algorithm (**Enum.**) and the variability-aware (**Our.**) algorithm to verify the FTA against the formula. For the SPL with the smallest set of features, it turns out that the product-based method always outperforms our new algorithm. Altogether, we observe an increase of 97.82% in verification time. This seemingly weakness is not due to the exploration of the state space itself, but rather to the time needed by the analysis of the feature models. According to additional benchmarks, this time amounts to 0.42 second on average. When the number of features and products increases, our variability-aware algorithm overcomes this overhead and outmatches the product-based one. For the medium-sized and big-sized SPLs, it decreases the total verification time by 53.09% and 81.71%, respectively. These results are highly encouraging as they show that our approach seems to be

Table 7.2: Verification times (in seconds, two decimals).

	6 features 16 products		10 features 72 products		13 features 512 products	
Property	**Enum.**	**Our.**	**Enum.**	**Our.**	**Enum.**	**Our.**
#1	0.35	0.58	3.33	0.98	10.52	1.04
#2	0.69	1.04	6.24	2.13	18.14	2.58
#3	0.44	0.71	5.17	2.67	13.13	2.72
#4	0.42	0.67	3.88	1.12	12.55	1.73
#5	0.39	0.66	4.07	1.01	9.75	1.04
#6	0.35	0.51	3.16	0.59	6.33	0.59
#7	0.66	0.93	5.57	1.97	18.86	2.15
#8	0.43	0.78	5.66	1.31	14.31	1.41
#9	1.06	1.48	9.05	3.59	31.99	3.93
#10	0.67	1.19	6.32	2.44	21.18	3.28
#11	0.69	3.67	7.55	10.98	14.60	11.69
#12	0.26	0.46	2.37	0.47	7.03	0.47
Total	**6.41**	**12.68**	**62.37**	**29.26**	**178.39**	**32.63**

more scalable with respect to the size of the SPL. For one formula, the product-by-product approach is a bit faster than ours in the small-sized and medium-sized cases. This formula is particular since all the features that have an impact on time lead to different path during the exploration. As a consequence, more zones have to be explored and compared during the verification process.

Threats to validity. Two characteristics of our experiments may harm the validity of our conclusions regarding relative efficiency of our algorithms. First, our evaluation is based on a sole small case study. To assess our approach at a greater scale, we should carry out experiments on additional examples. Second, the product-based method is evaluated through the classical TA checking algorithms implemented in our tool. There exist alternative solutions like UPPAAL which implement optimisations that our tool does not. However, the overhead of real-time models verification is mostly due to the representation of time. Given that we reused the very efficient DBM library of UPPAAL, we should draw the same conclusions even if we mapped the product-by-product method with existing tools.

7.6 Perspectives

The efficiency of FTA algorithms mostly depends on the data structures used to represent time and variability. On the above implementation, we

made use of separate encodings, *viz.* DBMs and BDDs, respectively. As alternatives, we could use a data structure that combines zone and binary variables representations. Clock Restriction Diagrams [160] and Constraint Matrix Diagrams [86] appear as promising candidates. Yet, to the best of our knowledge there is no implementation of these data structures which is available at a sufficient maturity level.

Another interesting perspective is the extension of real-time SPL modelling to non-Boolean features. This would allow one to directly express a real-time constraint in terms of feature attributes. As an example, the running speed of a processor may drastically influence the execution time of a software. Considering this extension would definitively impact on our model-checking procedure. Indeed, in this context we would not be able to use BDDs to encode sets of products. Second, we may not be able to translate the FTA into an infinite FTS anymore, since those cannot cope with non-Boolean features. There is thus a need to study the addition of non-Boolean features in FTS first. This is the focus of the next chapter.

Chapter 8

Non-Boolean Features

A fundamental limitation of existing SPL model-checking approaches, including FTS, is that they can deal with neither numeric features nor features appearing several times in a product, i.e. multi-features.[1] Numeric (as opposed to purely Boolean) features occur in FMs in the form of attributes associated to features. An example could be *"Maximum number of users"*, which can take different numeric values across the range of products. The multi-feature construct could be used to represent, e.g., processing units which number is variable from one product to another and which can be configured independently. A recent survey showed that engineers commonly need these constructs [115]. To transfer SPL model-checking techniques to industry, we thus have to support them. Despite evidence of their usefulness in practice, no SPL modelling tool currently supports multi-features [52], and SPL model checking tools support neither numeric features nor multi-features.

In this chapter, we propose a combined formalism that integrates FM with a behavioural specification language (*viz.* fPromela) to model the behaviour of SPLs with numeric attributes and multi-features. With the addition of attributes, optional behaviour can be made dependent not only on the presence or absence of features, but also on the satisfaction of arithmetic constraints over attributes. This implies a generalization of the underlying formalism, *viz.* FTS, and its associated model-checking algorithms.

8.1 Array-Based Semantics for Multi-Features

We first recapitulate the concept of multi-feature based on a running example. The CCSDS File Delivery Protocol (CFDP) is a highly-configurable deep-space file transfer protocol [61]. In the past, Boucher *et al.* [29, 113] helped Spacebel, a Belgian company, to develop an implementation of CFDP as an SPL [29]. The original CFDP FM has 98 features. Here, we consider

[1] Sometimes misleadingly referred to as *clones* in the literature [76, 142].

```
...                                                          1
typedef features {                                           2
    ...                                                      3
    bool Snd_min                                             4
}                                                            5
features CFDP;                                               6
active[2] proctype cfdp_entity(...) {                        7
    int i = 0;                                               8
        ...                                                  9
    if  ::  CFDP.Snd_min -> ... // send message              10
        ::  else -> ... // do nothing                        11
    fi;                                                      12
    ...                                                      13
}                                                            14
```

Figure 8.1: Partial CFDP model.

a small subset of the protocol, *i.e.,* the acknowledgement modes it offers. The corresponding sub-FM has 14 features and yields 1,058 different valid products. We had to limit ourselves to this subset because in addition to the variability, we had to model the behaviour of the features of the protocol, which is far more complex. We did that based on the protocol specification [61] and experimented with various SPL behavioural modelling languages. This turned out to be a difficult and time-consuming activity. The resulting models describe a communication scenario where an entity (e.g. a spacecraft) has to transfer a message to another one. Depending on the features of the protocol instance in each entity, properties like successful transmission may or may not be satisfied during the transaction. An illustrative excerpt of the fPromela model of the protocol is shown in Figure 8.1 (see Chapter 9 for more details about fPromela). It particularly focuses on the specification of the two entities exchanging messages. We use this excerpt further in this chapter to illustrate new concepts brought to fPromela. The FTS represented by the complete fPromela model has 1,812,652 states.

To better illustrate the concepts we introduce, we use TVL [52] as a textual representation for FMs (see more in Section 1.2). Figure 8.2 shows an excerpt of the TVL model of CFDP. The equivalent graphical representation appears on its right. In both representations, the FM's fundamental structure remains a tree that reflects the parent-child hierarchy between the features. At the top of the tree lies the *root* feature ($CFDP$). $CFDP$ has three children: *Entity*, *Message* and *Channel*. The group cardinality of $CFDP$ specifies that this feature must have exactly three children, whereas *Entity* must have one or two. Group cardinalities are not to be confused

with *feature cardinalities* [76, 142] which specify how many instances of a feature may exist in any given product.

The syntax and the semantics of attributes in TVL (and thus in FMs) are already defined [52] but the language does not support multi-features. To overcome this, we extended the syntax of TVL. For clarity, we name TVL* our new version of TVL. Before we give it a formal semantics, we have to deal with a number of issues. When no feature cardinality is given, as it is the case for all features in the model except *Entity*, the feature is implicitly assumed to occur at most once in each product. But whenever there is a need to allow a feature to have multiple instances, an explicit cardinality is added to the feature itself (as opposed to the group); we call such a feature a *multi-feature*. In the excerpt, the only multi-feature is *Entity*. It has a feature cardinality of exactly 2; hence two instances of this feature exist in each product. In our scenario, each instance corresponds to one of the two communicating spacecrafts. Defining *Entity* as a multi-feature allows the two spacecrafts to bear a different configuration. If *Entity* was a normal feature, the spacecrafts should necessarily be identical. In reality, it is more likely that they are not. Each instance of *Entity* must have at least one of the following child features: *Snd_min* (sending capabilities) and *Rcv_min* (receiving capabilities). Under *Snd_min* and *Rcv_min* lie additional features, which have been omitted in the figure. Features *Rcv_min* and *Message* both have an integer *attribute*. The attribute *timeout* determines the number of communication failures that are allowed before aborting a communication, while *size* models the number of data packets that must be sent for the transmission to end. Since there exist two instances of feature *Entity* and there also exist two instances of *RCV_min*, both of which has a distinct instance of attribute *timeout*. Feature *Channel* has an optional child feature *Reliable*, which specifies whether or not the communication channel is reliable.

8.1.1 Challenges in the Definition of Feature Cardinalities

As explained by Michel *et al.* [142], feature cardinalities can be given different semantics. When a non-terminal feature has a maximum cardinality greater than one, there are two possible interpretations of its group cardinalities. For instance, consider the FM in Figure 8.3a and the two instances shown in Figure 8.3b. According to the FM, feature A has two instances and a group cardinality of [1..2] (only one or two child features of A may exist). However, it is unclear whether the group cardinalities apply *locally* under *each* instance of A or *globally* for *all* instances of A. In other words, either each instance can have one or two child features (left diagram in Figure 8.3b), or there must be one or two child features of A altogether (right diagram). Since our approach is centred on multi-features, we consider the local interpretation like Michel *et al.* [142].

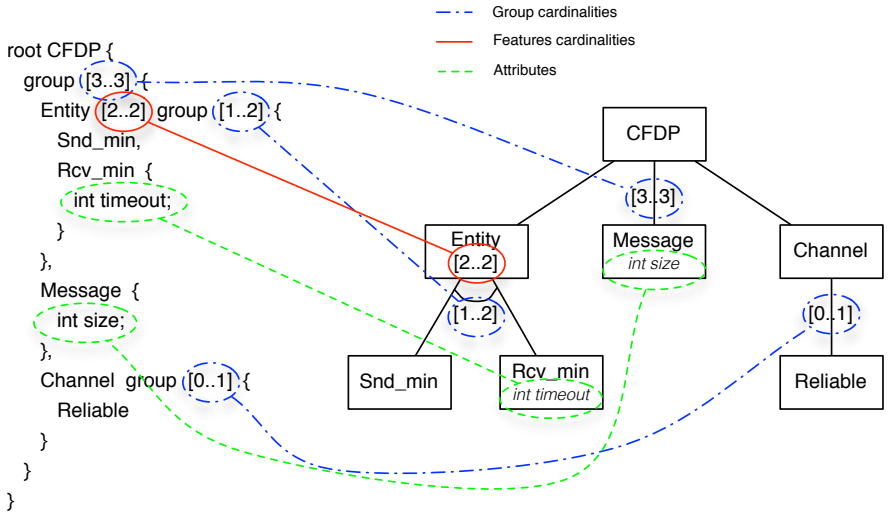
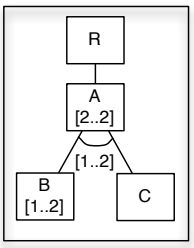


Figure 8.2: A subset of the CFDP feature model, shown in TVL and in a feature diagram.

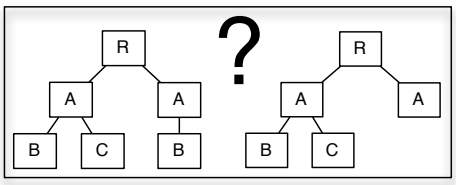
The second ambiguity lies in the scope of group cardinalities; see Figure 8.3c. In the left diagram, cardinalities restrict the number of *instances* of A's child features. On the contrary, in the right diagram, the two instances of B under the leftmost instance of A are counted once; in this case, cardinalities constrain the number of distinct child features. Like Michel *et al.* [142], we believe that the original intent of group cardinalities is to restrict the number of *distinct* features in a product; we thus consider the second option.

Multiple interpretations are also possible for feature cardinalities. Let us consider Figure 8.3d. As for group cardinalities, either feature cardinalities apply globally and count the total number of instances of a feature (left diagram), or they apply locally and count the number of instances of a feature under a specific instance of its parent feature (right diagram). As before, we adopt the most local option, *i.e.* the latter.

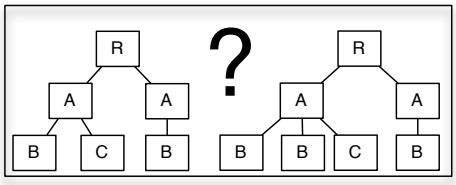
Another issue concerns the identification of a feature by means of its name. In FMs without multi-features, the (relative) name of a feature works as a unique reference to that feature. This is not the case for multi-features. Let us consider again the TVL* model of CFDP (Figure 8.2). The relative name *Snd_min* can refer to a child feature of either the first instance of *Entity* or the second one. We have to identify this instance using an "absolute" name (called *fully qualified*). In TVL, however, no construct exists for referring to a precise instance. Moreover, the semantics proposed by Michel *et al.* defines the children of a feature as a multiset of instances [142]. One



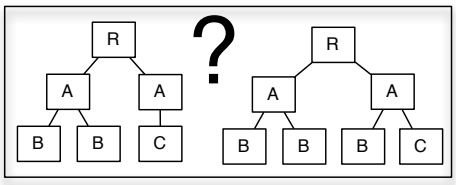
(a) FM example



(b) Group cardinalities (1)



(c) Group cardinalities (2)



(d) Feature cardinalities

Figure 8.3: Ambiguities introduced by feature cardinalities.

cannot refer to a precise element of a multiset. Our SPL behaviour specification language requires this capability; the definition of Michel *et al.* is thus inappropriate for our purpose.

We propose to represent the children of a feature by a set of arrays of instances. In a given array, all the instances are from the same feature. Each of them is identified by an index, the first index being zero. For example, the *Snd_min* child feature of the second instance of *Entity* is identified by the fully qualified name $CFDP[0].Entity[1].Snd\_min[0]$. When the maximum number of instances of a given feature is 1, the index 0 can be omitted; $CFDP.Entity[1].Snd\_min$ is thus equivalent to the above name. An attribute instance must be referred to using its fully qualified name as well, *e.g.*, $CFDP[0].Entity[1].Rcv\_min[0].timeout$.

Since we introduce multi-features in the syntax of TVL*, we have to provide means of specifying constraints over them, their attributes, and their number. Fully qualified names are already suitable to refer to precise instances or attributes. However, it is currently impossible to reason over a whole array. For example, one cannot express the requirement that *"the number of instances of a feature must not exceed the value of a given attribute of another feature"*, or that *"every instance of a feature must satisfy a given constraint"*. To address this limitation, we define the operator `card` which, given a fully qualified multi-feature name, returns its number of instances in the current configuration. E.g., `card(CFDP.Entity)` always returns 2. We also define two new types of constraints based on quantification: `forall(m)`$\{\phi\}$ and `exists(m)`$\{\phi\}$. Intuitively, they specify that for each (resp. at least one) instance of a multi-feature `m`, the *sub-tree* of this instance satisfies the constraint ϕ. For example, the constraint `forall` $(CFDP.Entity)$ $\{Snd\_min \vee Rcv\_min\}$ is satisfied if and only if every instance of *Entity* has at least one child. As we will see, a notion of *context* is required for defining the semantics of such formulae.

The last challenge is related to constraints over attributes. If a feature is not part of a product, references to its attributes point to an unknown value. In this case, it is undetermined how a constraint involving these attributes must be evaluated. Our new types of constraints already provide a solution to that problem by specifying constraints within the context of a instance. The evaluation of a constraint is thus performed under the assumption that the instance exists within its context.

A few attempts to reason on FM with attributes and multi-features exist. Czanercki *et al.* [76] define the semantics of an FM with feature cardinalities as the semantics of a context-free grammar derived from the FM. Although this work is already a major step forward, the operational style of the semantics creates accidental complexities. Michel *et al.* [142] defined another semantics based on multisets. This is already a major improvement. However, the issue with multisets lies in that elements of multisets do not have a unique reference, which impedes their use in a behaviour

specification language such as fPromela. Regarding additional constraints, Czarnecki and Kim [77] argue that OCL is a suitable language to express them. Even though we concur on this statement, we believe that the expressiveness of OCL is higher than needed for modelling these constraints, and that this excess reduces both readability and ease of use. Mazo *et al.* [140] use constraint logic programming to reason on FMs with multi-features and attributes. However, their procedure relies on a heavyweight transformation from visual FMs to XML, then to Gnu-Prolog's input syntax. Zhang *et al.* [168] propose a BDD-based approach for reasoning over multi-features, which inherently does not support attributes. A broader survey of feature cardinality analyses is found in [142]. The limitations of the above approaches lead us to define our own semantics for FMs with multi-features and attributes.

8.1.2 Extended Feature Models

Now that the new TVL* constructs have been informally introduced, we give them an abstract syntax and a formal semantics. More generally, the following definition remains valid for most feature modelling languages that support multi-features and attributes, and that form a tree structure.

Definition 8.1 (Extended feature model)
An FM model is a tuple* $(F, r, DE, \omega, \lambda, A, \rho, \tau, \Phi)$ *where:*

- F *is a non-empty, finite set of features.*
- $r \in F$ *is the root.*
- $DE \subseteq F \times F$ *is the decomposition relation between features, such that* $(f, g) \in DE \wedge (f', g) \in DE \Rightarrow f = f'$. *By* $children(f)$ *we denote the set of child features of* f.
- $\omega : F \to \mathbb{N} \times (\mathbb{N} \cup \{*\})$ *gives the cardinality of each feature. If* $\omega(f) = (n, m)$ *then* n *is the minimum cardinality of* f *and* m *its maximum cardinality. If* $m = *$ *then* m *can be any finite value higher than or equal to* n.
- $\lambda : F \to \mathbb{N} \times (\mathbb{N} \cup \{*\})$ *indicates the group cardinalities of a feature. It follows the same pattern as* ω.
- A *is the set of attributes.*
- $\rho : A \to F$ *is a total function that returns the feature declaring a given attribute. An attribute is thus declared by exactly one feature.*

- $\tau : A \to \{int, bool\}$ *assigns a type to each attribute.*
- Φ *is an additional constraint, i.e. a Boolean-valued expression over the features* F *and the attributes* A*.*

Furthermore, each FM must satisfy the following well-formedness rules:

- r *exists, is unique, and has no parent:* $\omega(r) = (1,1) \wedge \nexists f \in F \bullet (f,r) \in DE$*;*
- DE *is acyclic, that is,*

$$\nexists f_1, .., f_k \in F \bullet (f_k, f_1) \in DE \wedge \forall i \in \{1..k-1\} \bullet (f_i, f_{i+1}) \in DE.$$

The main difference between this new syntax and that of classical FMs lies in the definition of ω. Traditionally, ω was meant to identify optional features (see, e.g., the abstract syntax of TVL [52]; see also Section 1.2). Here, we make it more general since it defines the cardinality of a feature. Although this difference might seem thin at first sight, we already showed in Section 8.1.1 that it raises a number of important issues. The definition of Φ is also different here, as we have introduced attributes, a new operator and two new types of constraints. For convenience, we suppose that attributes are either integer or Boolean. Enumeration types can be mapped to integer and are thus implicitly supported too. The introduction of real attributes in the syntax is straightforward, but requires additional type checking.

Definition 8.2 (Additional constraints in extended FMs)
An additional constraint Φ *is a Boolean-valued expression over a set of features* F *and a set of attributes* A *formed according to the following grammar:*

$$\Phi ::= \top \mid f \mid a_b \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi_1 \mid V_1 \sim V_2 \mid exists(f)\{\Phi_1\} \mid forall(f)\{\Phi_1\}$$

where $f \in F$*,* $a_b \in A$ *is a Boolean attribute,* Φ_1 *and* Φ_2 *are additional constraints,* $\sim \in \{<, \leq, >, \geq\}$*, and* V_1 *and* V_2 *are integer-valued expressions. The latter are formed according to the following grammar:*

$$V ::= z \mid a_i \mid V_1 + V_2 \mid card(f)$$

where $z \in \mathbb{Z}$*,* $a_i \in A$ *is an integer attribute,* V_1 *and* V_2 *are integer-valued expressions, and* $f \in F$*.*

As explained in Section 8.1.1, only a fully qualified name can be used to identify a specific feature or attribute instance. Formally, it is a tuple of the form

$$\big((f_0, i_0), (f_1, i_1), \ldots, (f_k, i_k)\big) \in (F \times \mathbb{N})^{k+1}$$

or

$$\big((f_0, i_0), (f_1, i_1), \ldots, (f_k, i_k), a\big) \in (F \times \mathbb{N})^{k+1} \times A$$

respectively for a feature instance or an attribute instance. A fully qualified instance name must satisfy the following rules:

1. The first feature is the root indexed by 0: $(f_0, i_0) = (r, 0)$.
2. The decomposition hierarchy is respected:

$$\forall j \bullet 0 < j \leq k \bullet f_k \in children(f_{k-1}).$$

3. Each index satisfies the cardinalities of its associated feature:

$$\forall j \bullet 0 < j \leq k \bullet w(f_j) = (n, m) \Rightarrow 0 \leq i_j < m.$$

A fully qualified attribute name $\big((f_0, i_0), (f_1, i_1), \ldots, (f_k, i_k), a\big)$ is valid if and only if $(f_0, i_0), (f_1, i_1), \ldots, (f_k, i_k)$ is valid and a is an actual attribute of f_k: $f_k \in \rho(a)$. From now on, we assume the validity of every fully qualified name.

8.1.3 Formal Semantics

The purpose of FMs is to define the set of valid products in an SPL, i.e. the valid combinations of features and attribute values. We mentioned before that in SPLs without attributes and multi-features, a product is uniquely identified by a set of features. For reasons explained above, this representation is not appropriate when multi-features or attributes occur in the FM. Therefore we redefine a product as a couple $(\mathcal{F}, \mathcal{A})$ where $\mathcal{F} \subseteq (F \times \mathbb{N})^+$ is a set of feature instances identified by their fully qualified name, and $\mathcal{A} : (\mathcal{F} \times A) \to \mathbb{Z} \cup \{\top, \bot\}$ is a partial function that associates attribute instances with values. The validity of a product with respect to an extended feature model is defined as follows.

Definition 8.3 (Extended feature model semantics)
Let $fm = (F, r, DE, \omega, \lambda, A, \rho, \tau, \Phi)$ be an FM and $p = (\mathcal{F}, \mathcal{A})$ be a product. Then p is valid according to fm, noted $p \models fm$, if and only if, for any $(f_0, i_0) \ldots (f_k, i_k) \in \mathcal{F}$ we have:*

- *$\mathcal{F}$ contains the root: $(r, 0) \in \mathcal{F}$;*

- *$\mathcal{F}$ respects the decomposition hierarchy:*

$$\forall i \bullet 0 < i \leq k \bullet (f_{k-1}, f_k) \in DE;$$

- *every instance in $\mathcal{F}$ has its parent in $\mathcal{F}$ as well:*

$$(f_0, i_0) \dots (f_{k-1}, i_{k-1}) \in \mathcal{F};$$

- *the i-th instance of a feature exists only if the $(i-1)$-th does:*

$$i_k > 0 \Rightarrow (f_0, i_0) \dots (f_k, i_k - 1) \in \mathcal{F};$$

- *any attribute instance related to a given feature instance has a defined value if and only if the feature instance exists:*

$$\big((f_0, i_0) \dots (f_k, i_k), a\big) \in dom(\mathcal{A}) \Leftrightarrow (f_0, i_0) \dots (f_k, i_k) \in \mathcal{F};$$

- *$\mathcal{F}$ satisfies the feature cardinalities:*

$$\omega(f_k) = (n, m) \Rightarrow \big(i_k < m \wedge n > 0 \Rightarrow (f_0, i_0) \dots (f_k, n-1) \in \mathcal{F}\big);$$

- *$\mathcal{F}$ satisfies the group cardinalities:*

$$\lambda(f_k) = (n, m) \Rightarrow n \leq |\{(f_0, i_0) \dots (f_k, i_k)(f_{k+1}, 0) \in \mathcal{F}\}| \leq m;$$

- *p satisfies the additional constraints Φ.*

Note that we consider that instances are contiguously placed in arrays, which simplifies the verification of multi-features and constraints over them. For instance, in the above validity rule for group cardinalities, a child feature is counted if and only if its instance of index 0 is enabled. As explained above, the constraints of the form `forall(m)`$\{\phi\}$ and `exists(m)`$\{\phi\}$ introduce the notion of *context*, *i.e.*, a sub-tree of the model. For the context to be well-defined, it is sufficient to know the instance it refers to.

Definition 8.4 (Feature context)
Let $fm = (F, r, DE, \omega, \lambda, A, \rho, \tau, \Phi)$ be an FM. Then $(f_0, i_0) \dots (f_k, i_k) \in (F \times \mathbb{N})^+$ is a context of fm if and only if it is a valid fully qualified feature instance name.*

The initial context is the root feature. A valid product $p = (\mathcal{F}, \mathcal{A})$ must thus satisfy the additional constraints Φ in the context, noted $(r, 0)$: $p \models_{(r,0)} \Phi$. The satisfiability rules are given as follows.

Definition 8.5 (Satisfiability of additional constraints)
Let $p = (\mathcal{F}, \mathcal{A})$ be a product. Then the satisfiability of an additional constraint by p in a context c is determined according to the following rules:

$$
\begin{aligned}
(\mathcal{F}, \mathcal{A}) \models_c \top &\Leftrightarrow \top \\
(\mathcal{F}, \mathcal{A}) \models_c f &\Leftrightarrow \exists (f_0, i_0) \dots (f_k, i_k) \bullet \\
&\qquad c(f_0, i_0) \dots (f_k, i_k)(f, 0) \in \mathcal{F} \\
(\mathcal{F}, \mathcal{A}) \models_c a_b &\Leftrightarrow \exists (f_0, i_0) \dots (f_k, i_k) \bullet \\
&\qquad \mathcal{A}(c(f_0, i_0) \dots (f_k, i_k), a_b) = \top \\
(\mathcal{F}, \mathcal{A}) \models_c \Phi_1 \wedge \Phi_2 &\Leftrightarrow ((\mathcal{F}, \mathcal{A}) \models_c \Phi_1) \wedge (\mathcal{F}, \mathcal{A}) \models_c \Phi_2 \\
(\mathcal{F}, \mathcal{A}) \models_c V_1 \sim V_2 &\Leftrightarrow [\![V_1]\!]_c \sim [\![V_2]\!]_c \\
(\mathcal{F}, \mathcal{A}) \models_c \neg\Phi &\Leftrightarrow \neg((\mathcal{F}, \mathcal{A}) \models_c \Phi) \\
(\mathcal{F}, \mathcal{A}) \models_c forall(m)\{\Phi\} &\Leftrightarrow \forall i \bullet (c, (m, i)) \in \mathcal{F} \bullet (\mathcal{F}, \mathcal{A}) \models_{c,(m,i)} \Phi \\
(\mathcal{F}, \mathcal{A}) \models_c exists(m)\{\Phi\} &\Leftrightarrow (c \in \mathcal{F} \Rightarrow \exists c, (m, i) \in \mathcal{F} \bullet (\mathcal{F}, \mathcal{A}) \models_{c,(m,i)} \Phi)
\end{aligned}
$$

where $\sim \in \{<, \leq, \geq, >\}$ and $[\![V]\!]_c$ is the valuation of the integer-valued expression V within the context c, that is,

$$
\begin{aligned}
[\![n]\!]_c &= n \\
[\![a_i]\!]_c &= \mathcal{A}(c(f_0, i_0) \dots (f_k, i_k) a_i) \\
[\![V_1 + V_2]\!]_c &= [\![V_1]\!]_c + [\![V_2]\!]_c \\
[\![card(f)]\!]_c &= |\{c(f_0, i_0) \dots (f_k, i_k)(f, i) \in \mathcal{F}\}|
\end{aligned}
$$

In the above rules, c merely acts as a prefix for feature name. References to multi-features can occur in constraints, and express that at least one instance of the feature exists. Given our convention about array contiguity, this boils down to requiring the presence of the instance at index 0: p $\models_c$ m $\Leftrightarrow (c, (m, 0)) \in \mathcal{F}$. As in TVL [52], the semantics of some constraints depend on the evaluation of arithmetic expressions, variables, or operators. For our new *card* operator, the evaluation is defined as the size of a set. If it is compared to a constant value v then the constraint can be reduced to

a Boolean form:

$$p \models_c card(f) \geq v \Leftrightarrow \exists (f_0, i_0) \dots (f_k, i_k) \bullet \tag{8.1}$$
$$c(f_0, i_0) \dots (f_k, i_k)(f, v-1) \in \mathcal{F} \tag{8.2}$$
$$p \models_c card(f) > v \Leftrightarrow \exists (f_0, i_0) \dots (f_k, i_k) \bullet \tag{8.3}$$
$$c(f_0, i_0) \dots (f_k, i_k)(f, v) \in \mathcal{F}. \tag{8.4}$$

We will also make use of this property to reduce the cost of verifying these constraints in behavioural specifications.

8.2 On the Specification of Non-Boolean SPLs

The semantics of FM* allows us to determine which combinations of features constitute valid products. Still, we need a way to express the behaviour of these products in a concise (that is, not individual) manner. Based on the syntax of TVL*, we generalize fPromela with feature expressions supporting multi-features and attributes. Then we can verify models of this language against properties using ProVeLines. We name our extension fPromela*. It extends fPromela's syntax with more general data types for representing features. In fPromela, features are declared as Boolean fields of a specific data structure named `features`. For multi-features, this representation is not sufficient; they have to be declared in a dedicated user-defined data structure. For example, the data structure defined for the *Entity* feature would be:

```
typedef _Entity {                1
   bool is_in;                   2
   bool Snd_min;                 3
   bool Rcv_min                  4
};                               5
```

The first field of a structure is a Boolean called `is_in`. Its truth value defines the presence or the absence of a feature instance. The structures can be nested. Hence if, say, *Snd_min* is not a terminal feature, we can declare it as an instance of another type of structure. Attributes are declared as fields of the attribute type. More generally, there exists a transformation from an FM* to a set of fPromela* data structures. Each data structure represents a non-terminal feature, and contains one field per attribute and child feature. The type of the latter is either Boolean or another data structure, respectively if the feature is terminal or not. When the maximum cardinality of a feature is $1 < m < \infty$, the feature must be declared as an m-sized array of its corresponding type:

```
typedef features {               1
```

```
    _Entity Entity[2];                    2
    int timeout;                          3
    ...                                   4
};                                        5
```

As shown in the above example, feature attributes are straightforwardly encoded as a field of the corresponding type.

Since fPromela does not support arrays of an unknown size, the transformation cannot handle infinite maximum cardinality (represented as a $*$ in TVL*). However, we may assume that in real cases there always exists a realistic upper bound. Unlike in fPromela, the use of nested user-defined structures is compulsory in fPromela*. Indeed, a fully qualified name is required to refer to a precise instance (see Section 8.1.1). fPromela* offers a simple way to represent valid fully qualified names: references to fields of the corresponding user-defined structures, starting from that of the root feature. For example, the *Snd_min* child feature of the second instance of *Entity* is uniquely referred to by `CFDP.Entity[1].Snd_min`.

fPromela* generalizes feature expressions as well. We distinguish between *static* and *dynamic* formulae. The former specify one of the following requirements:

1. the presence or absence of a given instance (e.g., `CFDP.Entity[1].Snd_min`);
2. constraints over attributes (e.g., `CFDP.Message.size > 0`);
3. constraints over number of instances (e.g., `card(CFDP.Entity)==2`).

As in TVL*, the third type of constraints comes down to the first one when the *card* operator is compared to integers (see Equation 8.1). We can determine the products able to execute a transition associated with a static formula without executing the fPromela* model.

On the contrary, dynamic formulae are evaluated at runtime. Typically, they define constraints over attributes and number of instances in terms of variable values. For example, one could analyse the content of a CFDP message by iterating on its size:

```
i = 0;                                    1
do  ::  CFDP.Message.size > i             2
          -> ... i++;                     3
    ::  else -> break;                    4
od;                                       5
```

Note that for a given product, the actual size of the above model (in terms of states) depends on the value of the attribute in the product. Attributes in fPromela* are thus the key for specifying variable-size models. Instead

of attributes, we could also use the number of instances of a feature. In Section 8.3, we show that it is more efficient to verify a variable-size model rather than a set of fixed-size models, i.e. one per possible size.

Let us illustrate the last addition of fPromela* by means of the CFDP model. As mentioned before, two processes are declared and model the behaviour of two spacecrafts. These may have different configurations. In the partial model shown in Figure 8.1, however, we find but an ambiguous reference to feature *Snd_min*. We have seen that the introduction of multi-features permits the distinction between the features of the two entities. Still, we need a way to associate each process with the corresponding instance of *Entity*. To that aim, we define the feature *context* of a process as an instance of a feature. We propose two constructs to associate a context to a process. The first specifies it explicitly in the `run` statement of fPromela, which is used to dynamically instantiate a new process:

```
run [CFDP.Entity[1]] cfdp_entity(...)                    1
```

The second method consists in declaring that the number of instances of a process is equal to the number of instances of a specific feature:

```
active[card(CFDP.Entity)] cfdp_entity(...)               1
```

In this case, the context of each instance of `cfdp_entity` is a distinct instance of *Entity*. A process cannot exist outside its associated context. Accordingly, the first transition of a process is implicitly constrained by the feature expression encoding its context. Once defined, contexts can act as a prefix in feature expressions. For this purpose, one may use the keyword `this`, which points to the context of the process:

```
if   ::  this.Snd_min -> ...                             1
     ::  else -> ...                                     2
fi;                                                      3
```

References to features outside the context are still possible by using fully qualified names.

Originally, the semantics of an fPromela model is defined as an FTS. Since apart from feature expressions, fPromela* does not differ from fPromela, its semantics can be defined in terms of FTS where feature expressions have been generalized. We call the resulting formalism FTS*.

8.3 Evaluation

Since the semantics of an fPromela* model is an FTS*, we can reuse algorithms based on this formalism. We briefly describe how we implemented

the approach on top of ProVeLines, which we then used to carry out experiments. A complete description of the implementation is found in Chapter 9.

8.3.1 Implementation

No change to the verification algorithms themselves is required since these are independent of the representation of feature expressions. On the contrary, we updated the fPromela parser so that feature expressions are declared and referenced as user-defined data structures. We also relaxed syntactic constraints on them and modified the construction of featured program graphs accordingly. Dynamic formulae are handled by the semantic engine, i.e. the component that builds on the fly the FTS corresponding to the input fPromela model.

Given that more general feature formulae are allowed, the modules responsible for representing and verifying feature expressions are not appropriate anymore. Previously, these modules were based on either Binary Decision Diagrams (BDDs) [32] and their implementation in the CUDD library[2], or the MiniSat satisfiability solver [85]. BDDs have proven an improved efficiency in SNIP [53]; we therefore still use them in our evaluation. Yet, BDDs and SAT solvers alike cannot represent arithmetic constraints. Multi-features alone are not a problem in this regard but feature attributes are not necessarily Boolean. Satisfiability Modulo Theory (SMT) solvers appear as natural candidates for arithmetic constraints checking. Accordingly, our new feature expression modules are linked to Microsoft's Z3 [80]. Z3 implements many advanced techniques and structures for SMT solving. Equipped with Z3, ProVeLines is able to check the satisfiability of formulae over both the features and their attributes. However, this comes at the price of a significant reduction in efficiency, as further experiments reveal. Consequently, we developed two variants of the module: one equipped with CUDD and the other with Z3; the former supports only multi-features whereas the latter handles attributes as well. They are now included in the ProVeLines tool suite (see more in Chapter 9).

8.3.2 Overhead Measurement

We carried out experiments to evaluate the benefits of our approach, and to quantify the overhead that results from the additional verifications (see above) and/or the use of Z3 in place of BDD-based satisfiability solvers. The following tools are involved in the experiments: ProVeLines (as described in [53]), our BDD-based variant that supports multi-features (PVL-MF), and our second variant equipped with Z3 (PVL-Z3).

The addition of multi-features to fPromela* generates additional computations in the semantic engine of ProVeLines. Moreover, feature attributes

[2] vlsi.colorado.edu/~fabio/CUDD

require more general satisfiability-checking techniques, i.e. SMT solvers. These two generalizations are likely to negatively impact the performance of our tool. To quantify the resulting loss, we compare the time needed to model check three fPromela models against five properties each. The models include neither multi-features nor attributes. They are:

1. a minepump system [126, 59] (250,561 states to explore);
2. a restricted version of CFDP where attributes have received a fixed value and the two instances of *Entity* are regarded as a single feature (1,801,581 states to explore);
3. an elevator model inspired from Plath and Ryan [147] (58,945,690 states to explore).

We checked different kinds of properties including deadlock freedom, safety properties, and liveness properties. All benchmarks were run on a MacBook Pro with a 2,8 GHz Intel Core i7 processor and 8 GB of DDR3 RAM. We coded an automated script to execute them. To avoid random variations, we repeated each experiment several times and computed the average.

Results are shown in Table 8.1. We observe that multi-feature support in our BDD-based tool generates but a small increase in verification time (between 2% and 18% for minepump; between 0% and 5% for CFDP; between 1% and 12% for elevator). This increment does not depend on the size of the model. In every model, the increase is less than 6% for the longest-to-check properties. This indicates that the size of the property has no impact on the loss of performance either.

On the contrary, the Z3 variant suffers from a large performance drop. In minepump and CFDP, the checking time ranges from 21 to 31 times more than for the other two tools. The tool performs even worse on elevator: the verification time is multiplied by a factor between 468 and 962. For the first and fifth properties of that model, we could not even get accurate results; according to our estimations, verifying those properties would take 24 and 15 hours, respectively. Our SMT-based tool does not scale well with the size of the model.

We carried out further analyses to explain this phenomenon. For each tool, we computed the time required for satisfiability checking and formulae manipulation (i.e. conjunction, disjunction, negation, simplification) during each experiment. In Table 8.2 we provide the time share of the two computations in percentage, for every model and property.

For ProVeLines and PVL-MF, the absolute time needed for satisfiability checking (`Sat`) and formulae manipulation (`Man`) are identical. Indeed, given that all the models are free from multi-feature and attribute, the only difference between the two tools are the detection of dynamic formulae, which does not depend on `Sat` or `Man`. For these tools, it turns out that `Sat` constitutes but a small part of the total time (between 6% and 15%). Although

Table 8.1: Verification times (in seconds).

	ProVeLines	**PVL-MF**	**PVL-Z3**
minepump #1	2.72	3.21 (+18%)	78.22 (+2,776%)
minepump #2	2.88	2.95 (+2%)	67.86 (+2,256%)
minepump #3	4.90	5.47 (+12%)	88.63 (+1,709%)
minepump #4	2.96	3.42 (+16%)	65.72 (+2,120%)
minepump #5	9.22	9.76 (+6%)	197.43 (+2,041%)
CFDP #1	10.96	11.00 (+1%)	282.47 (+2,477%)
CFDP #2	2.23	2.41 (+2%)	63.37 (+2,741%)
CFDP #3	1.11	1.11 (+0%)	25.62 (+2,208%)
CFDP #4	3.82	4.00 (+5%)	113.36 (+2,867%)
CFDP #5	17.58	18.52 (+5%)	548.98 (+3,023%)
elevator #1	286.50	301.10 (+5%)	TIMEOUT
elevator #2	9.53	10.47 (+10%)	9,172.05 (+96,144%)
elevator #3	7.07	7.90 (+12%)	3,316 (+46,799%)
elevator #4	3.5	3.55 (+1%)	1,858.65 (+53,004%)
elevator #5	235.67	252.94 (+7%)	TIMEOUT

this share tends to slightly increase with the size of the model, it is not a major threat to scalability. On the contrary, `Man` is an increasingly important part of the total time. While the time share does not grow significantly between minepump and CFDP, it nearly doubles in the elevator case, ranging from 38% to 53%. This indicates that *managing a large number of BDDs is the main challenge for efficient BDD-based tools.*

In the third tool, `Sat` is the most costly computation. Its share is already between 73% and 82% for minepump and CFDP, and reaches astonishing proportions in the elevator case (about 95%). This clearly shows that expensive satisfiability checking threatens scalability. We measured the average time for a `Sat` computation. It amounts to $6.59 * 10^{-4}$ seconds for PVL-Z3 as opposed to $3.36 * 10^{-7}$ in BDD-based tools. In every model and property, the remaining computation time is due to feature manipulations for the most part. The average time of these is $1.62 * 10^{-5}$seconds – as opposed to $3.68 * 10^{-7}$ in the other tools. Together, `Sat` and `Man` always make up more than 92% of the overall verification time, and even over 95.5% in elevator.

In our context, SMT-based solutions thus appear less efficient than BDDs. This has to be confirmed through the replacement of Z3 by other SMT solvers. Indeed, Z3 offers many facilities that we do not need. Using another solver specifically designed for our purpose might yield better results. We leave that for future investigations. Nevertheless, the poor performance raises the need for alternatives to SMT.

Table 8.2: Share of verification time due to satisfiability checking and formula manipulation (in percentage).

	ProVeLines		**PVL-MF**		**PVL-Z3**	
	Sat	Man	Sat	Man	Sat	Man
M. #1	8.59	27.72	7.28	23.48	75.38	22.16
M. #2	6.07	18.26	5.93	17.83	80.37	16.82
M. #3	5.94	17.87	5.32	16.00	74.92	21.99
M. #4	6.02	18.18	5.21	15.74	80.35	17.77
M. #5	5.98	17.36	5.65	16.40	79.05	16.96
C. #1	10.14	28.79	10.10	28.69	73.07	20.86
C. #2	11.79	27.24	10.91	25.21	77.85	16.65
C. #3	8.44	20.77	8.44	20.77	77.14	15.96
C. #4	9.25	22.70	8.84	21.68	81.87	13.72
C. #5	9.38	24.32	8.91	23.09	78.57	13.49
E. #1	14.70	52.82	13.99	50.26	TIMEOUT	
E. #2	12.47	37.97	11.35	34.56	95.45	3.37
E. #3	12.09	38.23	10.82	34.21	94.68	4.87
E. #4	13.17	39.94	12.98	39.38	95.13	4.47
E. #5	11.31	37.57	10.54	35.0	TIMEOUT	

8.3.3 Explicit Attributes versus Boolean Conversion

As an alternative to Z3, we propose to transform integer attributes into a set of Boolean variables. For simplicity, we create one Boolean variable per attribute value. However, for attributes that can take n different values, only $\lceil log_2(n) \rceil$ Boolean variables are needed. This results in a model without attribute, which can thus be checked by PVL-MF. While this solution avoids the overhead of SMT solving, it reduces the performance of PVL-MF because more variables have to be dealt with.

The principles of this transformation applied to the CFDP model are the following. Each time a constraint over attributes is encountered in the model, it is first re-written in a form where only Boolean negation and greater operators occur. For instance, the constraint

$$CFDP.Message.size <= 3$$

is converted into

$$\neg(CFDP.Message.size > 3).$$

Then, we replace each constraint involving the greater operator with equivalent Boolean expressions. When the attribute is compared to a constant, only one Boolean variable is needed to represent the constraint. For instance,

the expression $CFDP.Message.size > 3$ is replaced by a Boolean variable $CFDP.Message.size3$ which is true if and only if $CFDP.Message.size$ has a value strictly greater than 3. In the CFDP model on which our analysis is based, attributes are also compared to a variable instead of a constant, for instance in the termination condition of a loop. Let us consider, e.g., the following fPromela excerpt:

```
...                                                    1
   int i = 0;                                          2
   do  ::  CFDP.Message.size > i -> ... i++;           3
       ::  else -> break ;                             4
   od;                                                 5
   ...                                                 6
```

The number of iterations in the loop is equal to the value of $CFDP.Message.size$. If we transform the attribute into Boolean variables, then we have to execute the loop once if $CFDP.Message.size0$ is true, one more time if $CFDP.Message.size1$ is also true, and so on. The loop can thus be replaced by a series of if-then statements whose inner block of instructions is a duplication of the core of the loop:

```
...                                                    1
   int i = 0;                                          2
   if  ::  CFDP.Message.size0 -> ... i++;              3
       ::  else -> break;                              4
   fi;                                                 5
   if  ::  CFDP.Message.size1 -> ... i++;              6
       ::  else -> break;                              7
   fi;                                                 8
   ...                                                 9
```

The above transformation is applicable only if the attribute takes its value in a *finite* domain. Finally, attributes are also compared directly to variable values, as illustrated in the following excerpt:

```
...                                                    1
   if  ::  fd_received == CFDP.Message.size ->         2
      completed = true;
       ::  else -> completed = false;                  3
   fi;                                                 4
   ...                                                 5
```

In this case, we have to transform a condition into a series of conditions, i.e. once for each value that the variable and the attribute may take:

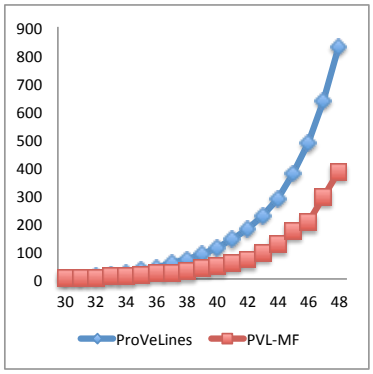
Table 8.3: Verification times for the CFDP example (in seconds).

	PVL-MF	**PVL-Z3**	**Speedup**
#1	9.79	898.88	91.82
#2	2.72	345.86	127.15
#3	1.35	137.05	101.51
#4	12.24	1694.61	138.45
#5	21.44	3661.89	170.79

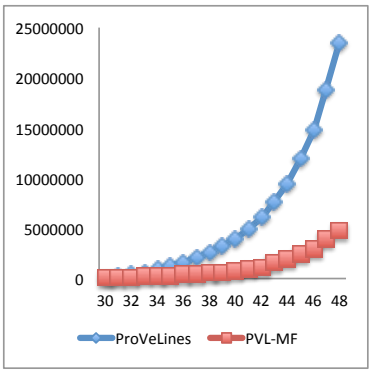
```
...                                                          1
   if  ::  fd_received == 1 && CFDP.Message.size1            2
      && !CFDP.message.size2
          -> completed = true;                               3
       ::  fd_received == 2 && CFDP.Message.size2            4
          && !CFDP.message.size3
          -> completed = true;                               5
       ...                                                   6
       ::  else -> completed = false;                        7
   fi;                                                       8
   ...                                                       9
```

As before, this transformation is applicable only to attribute and variable over a finite domain. By successively applying the above transformations each time an attribute appears, we end up with an equivalent fPromela model with only features and Boolean attributes.

The following experiments aim at evaluating whether this is more efficient than the Z3-based solution. We consider the complete CFDP model and compare the time needed by PVL-Z3 and PVL-MF to perform the verifications. Results are shown in Table 8.3. It turns out that the transformation implemented in PVL-MF yields far better results than PVL-Z3. We observe a speedup ranging from 91.82 to 170.79. Interestingly, apart from Property #1, the difference in speed increases as the properties are longer to check (see in particular Property #5). This corroborates our previous results and tends to show that in our context, attributes conversion is more viable than SMT. This has to be confirmed through additional experiments with more specific SMT solvers, though. Furthermore, this conversion may not always be as straightforward as for fPromela.



(a) Verification times (in sec.)



(b) Number of explored states

Figure 8.4: Benchmark results for the sieve of Eratosthenes.

8.3.4 Variable-Size Models

In Section 8.2, we mentioned that multi-features and attributes can represent variable-size models. Our new tools have thus the capability to verify in a single execution a set of models varying only by their size. They avoid redundant checking of common parts of these models, and are thus potentially more efficient. However, multi-feature support has an impact of performance. We must therefore evaluate if the avoidance of redundancy offsets the loss in efficiency.

We consider the sieve of Eratosthenes modelled in fPromela*. The size of the model is determined by the number of primes to compute. We compare the performance of (1) ProVeLines applied on models where the number of primes is fixed, and (2) PVL-MF executed on a model where this number is equal to the number of instances of a feature. For a maximum number n, we compute the time needed by ProVeLines for successively checking models with 1 to n prime numbers to compute. Then, we execute PVL-MF on the variable-size model.

Figure 8.4a shows the resulting verification times for n ranging from 30 to 48, whereas Figure 8.4b presents the number of explored states. Although the checking time always grows exponentially, it turns out that PVL-MF performs increasingly better than ProVeLines. As illustrated in Figure 8.4b, this is because the former explores fewer and fewer states than the latter. However, we observe that for PVL-MF, the verification time raises more rapidly than the number of explored states; the overhead due to multi-features has again a visible impact on the overall performance. Moreover, the benefit might not always exist. When the largest model has significantly more states than the smaller ones, the time needed to check these is negligible. In such cases, PVL-MF might not be more efficient than ProVeLines applied on each model size.

8.4 Perspectives

We dealt with feature cardinalities and numeric features in SPL model checking. Multi-features and feature attributes have received little support, be it in FMs or behavioural specification languages. These constructs pose many problems of both theoretical and technical nature. Nonetheless, the definition of languages supporting them is essential. Both constructs are useful in terms of expressiveness, conciseness, and readability. In a behaviour specification, they allow the definition of variable-size models. Our experiments showed that it is easier to verify such models than successively checking the same model with different sizes. Multi-features are also useful for specifying parallel processes that differ by their configuration. To deal with them, we introduced the notion of feature context for parallel processes. More generally, contexts can relate parallel process with *different* FMs. Combined with feature-model composition [2], this result opens the way for behavioural verification of SPL compositions. This is an interesting direction for future work.

Handling attributes in SPL model checking is difficult. SMT solvers appeared as natural candidates to represent and reason over numeric feature formulae. Moreover, they are increasingly used in model checking. They have proven their usefulness in bounded model checking and verification of infinite-state and array-based systems [93]. We showed that SPL model checking is another interesting application domain for SMT solvers. However, these have to be combined with heuristics in order to provide an acceptable level of performance in our context. However, our experiments with Microsoft's Z3 revealed that the overhead of SMT satisfiability checking is too important, which forced us to consider alternatives like converting attributes into Boolean variables. Yet, we need but a small subset of Z3's capabilities. Another solver optimised for our purpose could perform better. A complementary solution is not to check feature formulae each time it is needed. This leads to exploring more states than necessary but reduces the number of calls to the solver.

Moreover, Boolean conversion is not always practical. The behavioural models we considered do not include quantitative aspects like real-time or cost/rewards. Feature attributes in these contexts likely represent more complex forms of variability, *e.g.*, data throughput, processor speed, or energy consumption. Model checking such SPLs requires the combination of this work with quantitative formalisms like FTA (see Chapter 7). This is non-trivial; several theoretical issues like decidability are expected. On the technical side, the use of SMT solvers might be a mandatory step. Still, it constitutes an interesting problem that deserves to be further explored.

Part IV

Implementation

Chapter 9

ProVeLines: A PROduct Line of VErifiers for Software Product LINES

In this chapter, we discuss the implementation of all the theoretical developments presented so far. Although some of the earlier chapters have already shed some light on it, we give more details about the resulting tool, ProVeLines. Its origin is a tool named SNIP [53], which is the realization of the FTS theory as presented in Chapter 3. SNIP is one of the first SPL model-checking tool, along with fNuSMV [58, 54], SPLVerifier [12], and VMC [156]. In SNIP, SPL behaviour is modelled in fPromela (see Section 9.3). The tool implements semi-symbolic FTS algorithms, that is, based on an explicit representation of the state space and on a symbolic encoding of products into feature expressions. SNIP is linked to TVL, one of the latest feature modelling language. Basically, the model checker invokes an external library which extracts, from a TVL model, a feature expression encoding the valid products. Given an fPromela model and a TVL model, SNIP can analyse the behaviour of the valid products in search for deadlocks, failed assertions, or violations of LTL formulae. SNIP already constitutes a usable proof of concept for the efficient verification of SPLs. However, it implements only the state-of-the-art verification methods presented in Chapter 3, and thus lacks the theoretical developments we proposed.

A solution to implement our methods is to extend SNIP. Unfortunately, the different extensions of FTS we designed require different algorithms or data structures to represent and manipulate efficiently all the new constructs. For instance, the inclusion of real-time in FTS changes the definition of the state space and the logic for specifying properties, whereas numeric features oblige the use of an SMT solver. As for abstraction, it brings slight modifications to the checking algorithms and the implementation of a refinement procedure on top of these. Moreover, all these extensions are not

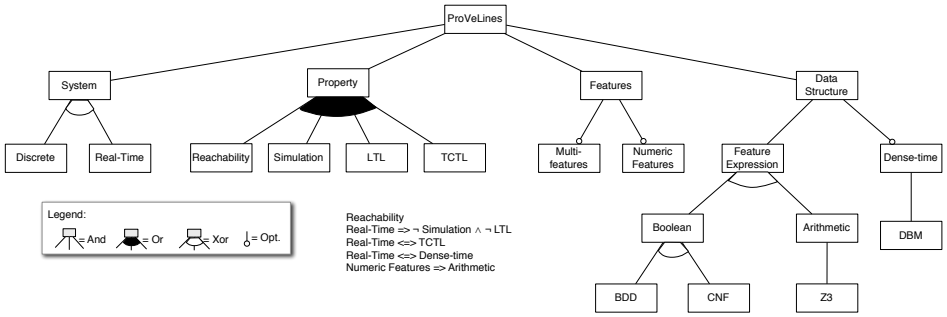


Figure 9.1: ProVeLines' FM

necessarily compatible. Our abstraction methods rely on the preservation of LTL; they could thus not be applicable to TCTL.

Given this variability, it appears more efficient to design, for each subset of extensions, a new variant of the tool that manipulates only the required constructs. These variants share many commonalities in source code, architecture, specification language, and algorithms. Moreover, new variants could be developed to implement future FTS-based approaches. The need for managing all the existing and future variants motivated us to re-engineer SNIP as a product line named ProVeLines, of which SNIP is but one of the products. The tool variants are built from the same code base by pruning code and setting compilation options. ProVeLines is the first SPL model-checking toolset to provide all the aforementioned functionalities. It is open-source and freely available on our website.[1]

9.1 ProVeLines' Variability

Developing ProVeLines as an SPL allows us to tailor it to the specific needs of the user. Indeed, many of its features augment its capabilities but increase the verification time. In a specific variant, the unneeded features should thus be removed. We organize the features of ProVeLines in an FM shown in Figure 9.1. Variability within ProVeLines originates from four factors: the model of the `System`, the `Property`, the supported types of `Features`, and the `Data Structures`. Altogether, ProVeLines consists of 40 unique products, including SNIP.

ProVeLines can check `Discrete` or `Real-Time` models. To support the latter, modifications are required in the specification language (*viz.* to handle timed statements in fPromela), the underlying formalism (to encode timed statement into real-time data structures), and the verifier (to make the verification aware of real-time, that is, to consider FTA instead of

[1] https://projects.info.unamur.be/fts/provelines/

FTS). ProVeLines can perform four types of verifications: `Reachability`, F-`Simulation`, `LTL` checking, and `TCTL` checking. When real-time models are considered, ProVeLines is limited to reachability and TCTL. First, we have not studied F-Simulation for real-time models. Second, LTL cannot reason on real-time, and the satisfiability problem for its real-time extension, named Metric Temporal Logic [123], is undecidable [8]. Accordingly, feature `Real-time` cannot coexist with features `LTL` and `Simulation`. On the contrary, TCTL can be enabled only if real-time models are considered. All the verification algorithms are implemented as part of the *checker* component.

In addition to Boolean features, ProVeLines supports `Multi-Features` and `Numeric Features`. As features have to be declared in the fPromela model, the support for non-Boolean features requires modifications to the language. The components dealing with variability have to be modified as well since the data structures that encode feature expressions depend on the types of considered features. In our implementation, variability related to system types, algorithms, and features is implemented in the form of preprocessor directives, e.g., *#ifdef*.

ProVeLines provides three data structures to represent feature expressions and check their satisfiability. For Boolean features, ProVeLines can use either Binary Decision Diagrams (`BDD`) [32] (implemented in the CuDD[2] library) or formulae in Conjunctive Normal Form (`CNF`) checked by MiniSat [85]. As it was already the case in SNIP [53], BDDs are more efficient than CNFs, whose size rapidly grows as more features are added. Yet, further empirical studies are needed to confirm this observation. To handle numeric features, we developed a wrapper to Microsoft's `Z3` [80]. In `Z3`, formulae are encoded in native data structures called abstract syntax trees. ProVeLines uses them to encode feature expressions and calls Z3's algorithms to check their satisfiability. The choice of a data structure is applied via compilation options. To encode real-time, ProVeLines relies on UPPAAL's implementation of Difference Bound Matrices (`DBM`) [23] combined with feature information. Observe that SNIP is the variant with features `Discrete`, `LTL`, `Reachability`, and `BDD`.

9.2 Overview of the Architecture

An overview of the architecture of ProVeLines is given in Figure 9.2. The core of our tool consists of four main components: *fts*, *features*, *fm* and *checker*. *FTS* is an interface providing a set of methods to access the different elements of the FTS: the states, their atomic properties, their transitions, etc. An actual implementation of this component is a bridge between a given behaviour modelling language and its semantics expressed in terms of FTS. It is, for instance, responsible for transforming on-the-fly an fPromela model

[2] http://vlsi.colorado.edu/~fabio/CUDD/

into an equivalent FTS. In the case of real-time models, it also manages the clock zones, although it dedicates their computation to an external component. Adding a new language thus requires to develop a new implementation of the FTS interface. When LTL formulae are checked, additional interfaces provides the methods necessary to observe and manipulate the corresponding Büchi automaton, its states, and its transitions.

Feature expressions are not handled in this component, which delegates this task to the *features* component. The latter offers methods to create, manipulate, and compute feature expressions. It also provides analysis operations (e.g., checking if a given expression encodes at least one valid product). It is linked to the *fm*, which consists of a wrapper to the TVL library [52] so as to extract a feature expression encoding the set of valid products. This allows the verification algorithms to ignore invalid products, thereby lessening the number of executions to analyse. ProVeLines could be linked to another variability modelling language via another implementation of the FM interface. Finally, *checker* implements the verification algorithms. It makes use of the *FTS* component to explore the state space of a given automaton, and of the *Features* component to performs operation of feature expression (e.g., accumulating them during the exploration and checking inclusion when a state is revisited).

Outside the core architecture lies useful additional components. The *math* component provides a set of data structures and functions to represent and manipulate mathematical constructs. Most of them are dedicated to logical formulae (be they binary or arithmetic), but one also finds a wrapper to UPPAAL's DBM library [23], as well as a formula minimizer (for display purpose). The implementation of logical formulae consists of wrappers to external libraries and tools. *util* contains general-purpose data structures and functions such as linked lists, stacks, hash functions, hash tables, as well as a *parser* subcomponent that provides functions to extract featured program graphs from an fPromela model.

9.3 Input Languages

Within ProVeLines, SPL behaviour are defined in fPromela, an annotative and executable modelling language based on the SPIN model checker's input syntax [114]. Basically, an fPromela model describes the behaviour of a set of processes subject to variability. Each process is specified in a `proctype` structure. Within a process, executable statements are expressed using constructs inspired by imperative programming languages. A statement can be annotated with a feature expression, such that only the subset of products satisfying the formula are able to execute the statement. Let us consider the excerpt shown in Figure 9.3. Features are declared as Boolean fields of a user-defined structure called `features`. In this model, two processes

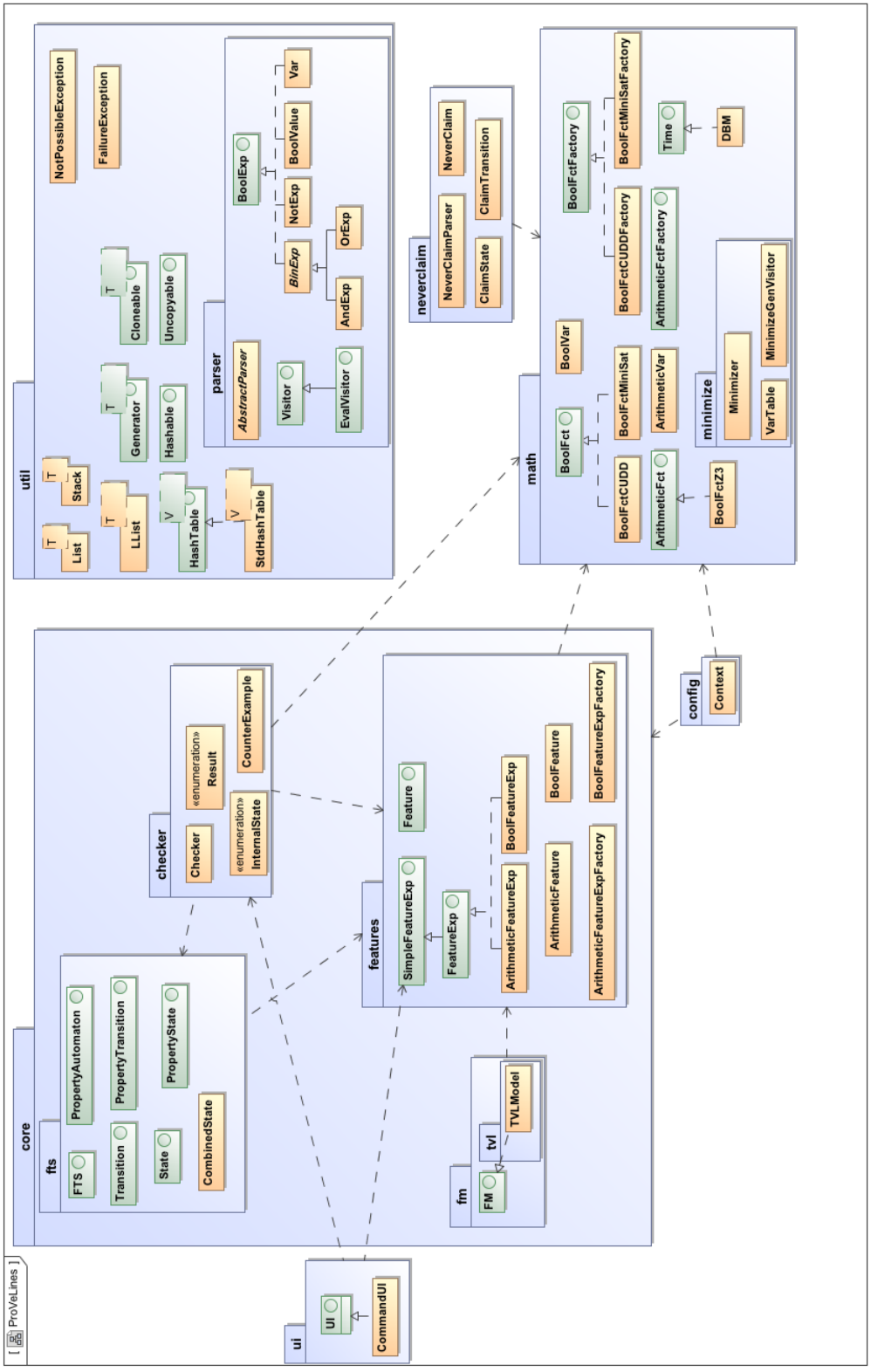


Figure 9.2: ProVeLines' Architecture

```
...
typedef features {
    ...
    bool Snd_min
}
features CFDP;
active[2] proctype cfdp_entity(...) {
    int i = 0;
        ...
    if  ::  CFDP.Snd_min -> ... // send message
        ::  else -> ... // do nothing
    fi;
    ...
}
```

Figure 9.3: An example of fPromela model.

of type `cfdp_entity` are specified. At some point, the specification of a `cfdp_entity` splits into two parts: one for the products satisfying the feature expression *Snd_min* and one for the others. The semantics of an fPromela model is an FTS. Each process is first translated into an FTS. A state corresponds to a variable valuation and a node of execution. Transitions between states are determined according to the executable statements. Feature expressions labelling transitions are directly derived from the fPromela model. Once all the processes have been translated, the final FTS is obtained by computing their parallel composition [53]. As more system types and kinds of features were supported, we extended the fPromela language with new constructs. In order to model SPLs with real-time behaviour, we incorporated timed statements in fPromela, which specify that the system needs time to execute the next action. Let us consider the following example:

```
typedef features {
   bool FastStart;
   bool FastStop
};
features f;
// clock declaration
clock c;
bool pumpOn = false;
active proctype toto() {
  off: pumpOn = false;
  // timed statements
       gd :: f.FastStart  ->
```

```
                while (c<7) wait;                    13
                when (c>4) reset(c) goto on;         14
            :: else          ->                      15
                while (c<10) wait;                   16
                when (c>7) reset(c) goto on;         17
        dg;                                          18
                                                     19
  on: ...                                            20
}                                                    21
```

We use real-time clocks to encode the passage of time (see Line 7). The statement *wait* specifies that the process can delay its next action as long as the value of a given clock remains in a certain interval (i.e. a *location invariant*). The statement `when` specifies that the process can trigger the next transition only when the value of a given clock lies in a certain interval (i.e. a *transition guard*). For example, Lines 14–15 specify that when feature `FastStart` is enabled the system needs between 4 and 7 time units to move from state `off` to state `on`. When this feature is disabled, the delay amounts to between 7 and 10 time units. We can check timed fPromela models, whose semantics is an FTA (see Chapter 7) against timed reachability properties such as *"for which products can the system be turned on in less than 5 time units"*. In this case, ProVeLines would find out that feature *FastStart* is needed to satisfy the property. Another recent extension of fPromela is the support for multi-features and numeric features. Since we already detailed the complete specification of these features in Section 8.2, we do not repeat it here.

9.4 User Interface

We now illustrate ProVeLines' command-line interface by means of a concrete example. As ProVeLines is based on SNIP, which is itself inspired by SPIN [114], the interface of both tools is similar [53]. We consider the SPL of minepump systems, which we used to carry out experiments in several previous chapters. For recall, this system consists of a controller, a pump, a water sensor, a methane sensor and a user. When activated, the controller should switch on the pump when the water level is high, but only if there is no methane in the mine. The model is composed of several communicating processes. We consider on the following property: *"There is never a situation in which the pump runs indefinitely even though there is methane."*; in LTL: $\neg\Diamond\Box(pumpOn \wedge methane)$. A call to ProVeLines to checking this property yields the following.

```
$ ./provelines -check -ltl '!<>[] (pumpOn && methane)' minepump.pml

Checking LTL property !<>[] (pumpOn && methane).
```

```
Property violated [explored 469 states, re-explored 0]
 - Products by which it is violated (as feature
 expression):
   (Start & Stop & MethaneQuery & MethaneAlarm & Low
   & High)

 - Stack trace:
[...]
```

The parameter `-check` specifies that we want to verify the fPromela model. Alternatively, we could use the parameter `-s` to print the featured program graphs of the fPromela model, or the parameter `-sim` to generate a random execution of the model. When `-check` is used, the parameter `-ltl` is followed by the LTL formula to verify. Using the above commands, the verification stops either when all the state space has been explored for all products, or as soon as a counterexample is discovered. As mentioned in Chapter 3, more than one counterexample might be needed, though. Therefore, an additional parameter, `-exhaustive`, makes the tool continue the verification until counterexamples are found for all violating products.

```
$ ./provelines -check -exhaustive -nt
  -ltl '!<>[] (pumpOn && methane)' minepump.pml

Checking LTL property !<>[] (pumpOn && methane).
Property violated [explored 469 states, re-explored 0]
 - Products by which it is violated (as feature
 expression):
   (Start & Stop & MethaneQuery & MethaneAlarm & Low
   & High)

[...]

Exhaustive search finished  [explored 13199 states,
re-explored 110745].
 - 16 problems were found covering the following
 products (others are ok):
(Start & High)
```

Note that we did not specify the feature model explicitly; in this case, ProVeLines automatically looks for a file named `minepump.tvl`. ProVeLines finds 16 violations and concludes that all products with features *Start* and *High* violate the property. This is not what we expected, as the property is supposed to be satisfied by all the products. Products without *Start* or *High* will never even start the pump, which is why they satisfy the property.

A look at the stack traces reveals a problem with the property. Basically, the controller has a central loop, in which it can receive three types of messages: user commands, methane alarm messages, and water level readings. The stack traces show in every case that the methane sensor sends an alarm message to the controller. However, as the choice of receiving one of the three messages is non-deterministic, the controller might ignore the alarm message indefinitely. In practice, such a behaviour is highly unlikely. It is thus reasonable to assume that the controller will infinitely often accept a message of each type if they are present. This assumption can be specified

as the LTL formula:

$$(\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel).$$

Then we obtain the following.

```
$ ./provelines -check -exhaustive -nt -ltl
  '([]<> read..) -> (!<>[] pump..)' minepump.pml

Checking LTL property ([]<> read..) -> (!<>[] pump..).
Property violated [explored 20674 states, re-explored
92326]
 - Products by which it is violated (as feature
 expression):
   (Start & Stop & MethaneQuery & !MethaneAlarm & Low
   & High)

[...]

Exhaustive search finished  [explored 26380 states,
re-explored 197637].
 - 8 problems were found covering the following
 products (others are ok):
(Start & !MethaneAlarm & High)
```

According to this result, feature *MethaneAlarm* is required to satisfy the property. This corresponds to what we expected, as feature *MethaneAlarm* alerts the controller of methane, leading it to shut off the pump. Normally, the example property is not expected to hold for products that do not have feature *MethaneAlarm*. It corresponds to a requirement implemented by the feature. We should therefore check it only against products that have the feature. This can be accomplished in ProVeLines using the `-filter` parameter. This parameter restricts the verification to a subset of all products specified as a feature expression (in TVL syntax). Limiting the previous check to products with *MethaneAlarm* yields the following.

```
$ ./provelines -check -exhaustive -nt
  -filter 'MethaneAlarm'
  -ltl '([]<> read..) -> (!<>[] pump..)' minepump.pml

Checking LTL property ([]<> read..) -> (!<>[] pump..).

Attention! Checks are only done for products
satisfying:
  MethaneAlarm!

Property satisfied [explored 21201 states, re-explored
188589].
```

The property is indeed satisfied by all relevant products.

When ProVeLines' `Real-Time` feature is enabled to check real-time SPLs, the commands to input are mostly the same. The sole difference lies in that TCTL formulae are checked as opposed to LTL. Therefore, the parameter `-tctl` replaces the previous `-ltl` parameter. In the following example, we determine which products of the real-time version of the minepump SPL can turn on in less than five time unit, that is, we check the TCTL formula

$\exists\Diamond_{<5} on$:

```
$ ./provelines -check -tctl 'EF (<5) on'
  minepump_rt.pml

Property violated
 - Products by which it is violated (as feature
 expression):
   (!FastStart)

 - Stack trace:
[...]
```

As mentioned in Chapter 7, only the products with feature *FastStart* can satisfy this property.

The last type of verification we may execute is simulation checking. Let us consider the following excerpt of fPromela model:

```
...                                              1
active proctype toto() {                         2
  int i = 0;                                     3
                                                 4
  gd :: f.Foo || f.Bar -> i++;                   5
     :: else -> skip;                            6
  dg;                                            7
                                                 8
  assert(i == 1);                                9
  ...                                            10
}                                                11
```

as well as an abstract version of this model:

```
...                                              1
active proctype toto() {                         2
  int i = 0;                                     3
                                                 4
  gd :: f.Foo -> i++;                            5
     :: else -> skip;                            6
  dg;                                            7
                                                 8
  assert(i == 1);                                9
  ...                                            10
}                                                11
```

This model abstracts feature *Bar*, which may change the behaviour of the products having this feature. If we run ProVeLines, we indeed obtain that the abstract model does not simulate the original model for the products having feature *Bar* but not feature *Foo*, and is thus not an appropriate abstraction:

```
$ ./provelines -check -tctl 'EF (<5) on'
$ ./provelines -sim -props p.prop
  orig.pml abstract.pml

Checking F-simulation.

Simulation holds for the following products:
  (!Foo & !Bar) | Foo
```

This result is not surprising: the abstraction given here is not an F-abstraction in the sense of Definition 6.2, since it does not satisfy Equation 6.1. Indeed, in the abstraction, the transition corresponding to the execution of the statement `i++` is available to fewer products (i.e. $[\![Foo]\!]$) than in the original model (i.e. $[\![Foo \vee Bar]\!]$). Conversely, the transition executing the statement `skip` is available to products $[\![\neg Foo \wedge \neg Bar]\!]$ in the original model, and to products $[\![\neg Foo]\!]$ in the abstract model.

9.5 Perspectives

We designed the architecture of ProVeLines in a way that makes it extensible. It can serve as a basis for the implementation of future FTS-based verification methods. In particular, we can broaden the portfolio of our checking algorithms, e.g., to enable the verification of stochastic and hybrid SPLs. As for real-time, the major challenge in these new approaches is to define efficient data structures to represent and manipulate quantitative data. The modular architecture of our tool facilitates the integration of other data structures, be it to represent feature expressions, real-time, or others. It thus enables a swift comparison between several alternative data structures.

We believe that the verification procedures implemented in ProVeLines have reached a sufficient maturity level to be confronted to real-world cases. The next challenge to face is to adapt the ergonomy of languages and tools so that engineers can actually use SPL model checkers without having a deep knowledge of the underlying theory. An unavoidable requirements to reach this goal is the integration of other high-level languages. Jeanjot [117] has already made a step in this direction, as he developed a prototype wrapper to Stateflow for ProVeLines. Willemart [161] went even further: he designed a textual language based on statecharts to represent variable behaviour, used this language to model a real-world SPL, and integrated it into ProVeLines to perform verifications on the model.

More generally, the low coupling between input and algorithms permits engineers to use, extend, and tailor ProVeLines even if they do not hold any expertise in model checking. The separation of concerns that drove ProVeLines development yield numerous facilities that can be used independently and in many contexts. Thereby, we hope that our tool will reach researchers

and practitioners alike. An interesting future work that would help achieving this objective is the development of a graphical user interface similar to that of UPPAAL [23], where users can draw models of their system, define properties to verify in an intuitive language, and observe the results graphically.

Part V

Applications to Other Formal Methods

Chapter 10

Modelling and Verification of Adaptive Systems

All the work presented so far aims at designing model-checking techniques and optimisations specifically for SPLs. In the next two chapters, we show that the ideas behind our approach can be applied to other automata-based formal methods. We focus first on providing a framework to model and verify adaptive systems.

Our society increasingly entrusts computerized systems with complex and critical tasks. These systems have to be adapted, or adapt themselves, to a rapidly evolving environment, while accomplishing their tasks reliably. Due to the short reaction times required, some of these adaptations have to be performed automatically, leading to *self-adaptive* systems (more simply called adaptive systems from now on). Such systems are usually structured in two levels. The base level manages the basic tasks of the system. It has a simple design that allows rapid response times, but does not allow to respond to exceptional conditions. For instance, a satellite control system is in charge of maintaining the attitude of the satellite so that the solar panels face the sun. It must react rapidly when the satellite starts to spin. On the other hand, the adaptive level can detect a change of conditions, and reprogram the base system to adapt to these new conditions. Again, the base system is efficient, but often not exhaustive. For instance, when entering the shadow of a planet, the system will be adapted to give priority to the orientation of the data transmission antenna.

The verification of adaptive systems is a topic that received a lot of attention from the scientific community. In their research roadmap for adaptive systems, Cheng *et al.* [42] stated that in the context of adaptive systems, the objective of quality assurance is to provide evidence that the system is able to cope with changes in its objectives and its environment. The classical validation and verification methods being meant for stable systems, there is a crucial need for novel techniques specific to adaptive systems. They presented

a framework for adaptive systems assurance, in which the system, the goals, and the context are subject to modifications. This results in a succession of models for the system and properties to verify. Following this idea, several verification methods model them as a set of programs [5, 125, 129, 167]. To ensure the satisfaction of intended properties in an unstable environment, the system is able to make transitions between those programs. The execution of such transitions is called an adaptation. By performing it, the system modifies its future behaviour. In this context, one distinguishes between local properties that specific programs must satisfy, global properties that must be satisfied by any execution of the system, and transitional properties that must hold during an adaptation. To specify the transitional properties, Zhang *et al.* [166] proposed a new logic called A-LTL. In their recent work [167], they provided an algorithm based on marking to verify an adaptive system against an A-LTL formula. Kulkarni *et al.* [129] consider that an adaptive system is a program able to add or remove components during runtime. Instead of traditional model checking, they use proof lattice as an alternative solution for verifying that all possible adaptations satisfy all the global properties. Adler *et al.* [3] propose a framework to model adaptive systems at a high level of abstraction. This framework allows one to perform model transformations to derive more abstract models and reduce verification complexity, e.g. through data abstraction. The authors also introduce algorithms dedicated to the verification of particular properties like stability and steadiness. There exist other verification approaches for adaptive systems that are based on model transformations [94, 18], but those are reduced to the verification of safety or invariant properties. Although they do not target model checking, Khakpour *et al.* [122] propose PobSAM, a language to model and specify adaptive systems and adaptation policies. The semantics of PobSAM is given in the form of a process algebra. This allows the authors to define behavioural equivalence modulo a new definition of bisimilarity. Instead of functional requirements, Filieri *et al.* [88] tackles the verification of non-functional requirements like reliability in the context of adaptive systems. For this purpose, they propose novel algorithms for checking efficiently parametric parametric discrete-time Markov chains.

In this work, we slightly derive from the original idea of representing adaptive systems as a set of programs. Instead, we propose to model adaptations by the notion of feature, borrowed from SPL engineering. Classically, a feature is an added functionality to the system, that responds to a (new) need of the customer. Here, a feature can also be an adaptation to environmental conditions. For uniformity, we also model such evolutions of the environment (in which we might include some parts of the system itself) as special features. They can be failures (in which case the associated behaviour describes the failure mode and effects), increase of power of an attacker (in which case the associated behaviour describes the modus operandi of attacks), etc. In some cases, preserving the functionality of the system is not

possible, e.g. in the presence of severe failures. Therefore the requirements need to allow for degraded functionality in such cases, and thus our requirements logic also should include dependency on features. For instance, when a solar panel is damaged, the satellite is allowed to shut down some of its non-prioritary facilities (e.g., observation of aurora borealis) to preserve its vital functions (e.g., avoiding falling on Earth) instead. We thus define *resilience* as the capacity to ensure such conditional requirements in presence of a changing environment (at end of Section 10.2). The resilience-checking problem departs from the classical model-checking problem in several ways, and thus requires specific adaptations of the classical algorithms, presented in Section 10.3.

10.1 Dynamic Software Product Lines

One way to model a dynamically adaptive software is to represent it as a set of static programs and transitions between them [167]. A transition between two programs models an adaptation of the system, which may be needed in case of changes in the properties of the environment. Since those programs are usually meant to satisfy the same set of high-level goals, they likely share commonalities in their structure and behaviour. Moreover, nowadays an increasing number of software systems are designed as product lines and we observe the emergence of *Dynamic Software Product Lines* (DSPLs) [97], especially in the mobile software industry. In order to cope with changing architectures and environments, these SPLs are equipped with the ability to change their set of features at runtime. In this context, we call *configuration* the set of features of a system at a particular point of time, and *reconfiguration* the process of altering the configuration of this system. The implementation details of reconfiguration are often hidden to the user who can only witness which *features* of the system have changed.

However, behavioural modelling approaches for SPLs like FTS do not consider that an SPL may have to adapt its behaviour due to unexpected changes in the environment. In other words, a given configuration is chosen and fixed through the whole execution of the system, which thus cannot react to changes in its environment. Here, we propose a new modelling formalism that allows dynamic reconfiguration. More precisely, we introduce *Adaptive Featured Transition Systems* (A-FTS), an extension of FTS meant for modelling DSPLs. A-FTS explicitly represent the variability of both the system and its environment. The latter is defined as a set of features, such that an *environment feature* is a Boolean characteristic of the environment that may change over time and that the software has the ability to perceive. The capability of the software to execute a transition thus depends on both its features and those of the environment. For the system, we distinguish between *fixed*, non-mutable features and *adaptable* features. To capture the

variability of both the system and the environment, we consider that a feature model is defined over a set of features $F = F_s \cup F_e$ such that F_s is the set of system's features, $F_a \subseteq F_s$ is the set of the system's adaptable features, and F_e is the set of features of the environment with $F_s \cap F_e = \emptyset$. According to the above, we define a system configuration (resp. an environment configuration) as a subset of F_s (resp. F_e), and a (complete) configuration is the union of these two.

An adaptable feature may be enabled or disabled at some points during the execution. Unexpected variations in the environment may force the software to adapt its configuration so that it still works properly according to intended requirements. Given that objective, we consider that after the system executes a transition, it is able to observe the features of the environment that are enabled at that time. According to the current configuration of the environment, the system may alter its own configuration so that it avoids failure or undesirable situations. Since we do not consider real-time constraints, we suppose that these reconfigurations are atomic and instantaneous. However, the system can be forbidden to reconfigure itself at some point because it must first terminate the execution of a given sequence of actions. In particular contexts, the environment can also be stable during a given period. The following formal definition of A-FTS takes all these considerations into account.

Definition 10.1 (Adaptive FTS)
An A-FTS is a tuple $(S, Act, I, AP, L, fm, \gamma)$ *where*

- S, Act, $I \subseteq S$, AP, *and* $L : S \to 2^{AP}$ *are defined as in Definition 3.2;*
- fm *is an FM over a set of features* $F = F_s \cup F_e$, *where* $F_s \cap F_e = \emptyset$ *and* $F_a \subseteq F_s$ *is the set of adaptable features;*
- $\gamma : S \times Act \times S \to (2^F \times 2^F \to \{\top, \bot\})$ *is a function that defines the transition relation.*

Note that unlike LTS and FTS, the transition relation is not defined as a set called $Trans$. Instead, we use the function γ to encode symbolically which transitions exist, which products can execute them and how the configuration of the system and the environment can evolve. More precisely, this function allows us to:

1. Determine whether of not the system can reach a state s' by executing an action α from a state s. If that is not the case then $\gamma(s, \alpha, s')$ is a function that returns $\bot$ whatever the configuration of the system and

of the environment. We also assume that for a given configuration, s' is unique:

$$\Big(\bigvee_{c',e'} \gamma(s,\alpha,s')(c\cup e, c'\cup e') \wedge \bigvee_{c'',e''} \gamma(s,\alpha,s'')(c\cup e, c''\cup e'')\Big) \Rightarrow s' = s''.$$

2. Restrict the set of configurations able to execute such a transition. Let us suppose that the system can reach s' from s by executing α if and only if its features satisfy the feature expression exp. For any system's configurations $c \in [\![exp]\!], c' \in 2^{(F_s)}$ and any environment's configurations $e, e' \in 2^{F_e}$, we have $\gamma(s,\alpha,s')(c\cup e, c'\cup e') = \top$. This definition of transition relation is thus more general than in FTS.

3. Restrain how the configuration of the system and the environment can evolve after the execution of the transition. For instance, let us suppose that if the system moves from s to s' by executing α while in configuration c then it cannot change its configuration at all. To express that constraint we define that for any system's configuration c' and environment's configurations e and e', we have that

$$c \neq c' \vee e \neq e' \Rightarrow \gamma(s,\alpha,s')(c\cup e, c'\cup e') = \bot\ .$$

Note that γ must be defined such that only the adaptable features of the system may be enabled or disabled during runtime:

$$(c \setminus c') \cup (c' \setminus c) \not\subseteq F_a \implies \neg\gamma(s,\alpha,s')(c\cup e, c'\cup e')$$

for any $s, \alpha, s', c, c', e, e'$. Moreover, any reconfiguration of the system and the environment must ensure that the new configuration is valid (that is, it must satisfy the constraints of the feature model). Accordingly, the function γ must be such that:

$$c'\cup e' \not\models fm \implies \gamma(s,\alpha,s')(c\cup e, c'\cup e') = \bot$$

for any $s, \alpha, s', c, c', e, e'$.

Example 10.1.1 *An example of an A-FTS is graphically illustrated in Figure 10.1. It depicts a small behavioural model of an adaptive routing protocol inspired by the case study of Zhang et al. [167]. The system has one adaptable feature encryption, which leads to two possible configurations. Similarly, the environment has one feature* $safe$ *that may or may not be enabled. Initially, the system is in state* `ready`*. Once it receives a message, it enters the state* `received`*. Then, it routes the message and enters either* `routed-safe` *or* `routed-unsafe` *depending on whether the environment is safe or not. This information is captured by the environment's feature* safe*. In our graphical representation, we model these restrictions by writing the feature expression*

that must be satisfied by the current configuration of the system and the environment for the transition to be executable. Formally, we make use of the transition relation γ for defining those restrictions. For instance, we know that the transition between `received` *and* `routed-unsafe` *cannot be executed in a safe environment; accordingly, we define γ such that*

$$\gamma(\texttt{received}, \texttt{route()}, \texttt{routed-unsafe})(c \cup e, c' \cup e') = \bot$$

where $safe \in e$ and for any e',c,c'. In order to increase readability, we do not explicitly represent the whole definition of the function γ. If the environment is safe then the system simply sends the message, reaches the state `sent-safe`*, and ends up going back to the state* `ready`*. If the environment is not safe and if the system's feature encryption is enabled, then the system encrypts the message and sends it afterwards. Otherwise, it sends the message unencrypted. In every case, the system eventually returns to state* `ready`*. In our graphical representation, the set of atomic propositions satisfied by a given state is written directly below it. Additionally, we define that once the system has routed the message it cannot change its configuration until it reaches state* `ready` *again. Similarly, we suppose that the features of the environment are stable between the routing and the sending. We capture those restrictions by means of the transition relation γ. For instance, we have*

$$\gamma(\texttt{routed-safe}, \texttt{send()}, \texttt{sent})(c \cup e, c' \cup e') = \top \Leftrightarrow (c = c' \wedge e = e')$$

for any c, c', e, e'. A property of interest for this system is that while the environment is unsafe, no package is sent before it is encrypted. Further in this chapter, we introduce a new logic able to express such properties.

As mentioned in Chapter 2, an execution of a transition system is defined as an infinite alternating sequence of "states" and actions. Unlike LTS and FTS, the concept of state in A-FTS does not only refer to the state of the system itself, but also to its configuration as well as that of the environment. In order to avoid ambiguity, we call that a *macrostate*.

Definition 10.2 (Macrostate)
A macrostate of an A-FTS is a triplet

$$(s, c, e) \subseteq S \times 2^{F_s} \times 2^{F_e}$$

such that $c \cup e \in [\![fm]\!]$.

For example, one of the macrostates of the A-FTS presented in Example 10.1.1 is $(ready, \emptyset, \{safe\})$. If the A-FTS is in this macrostate, it means

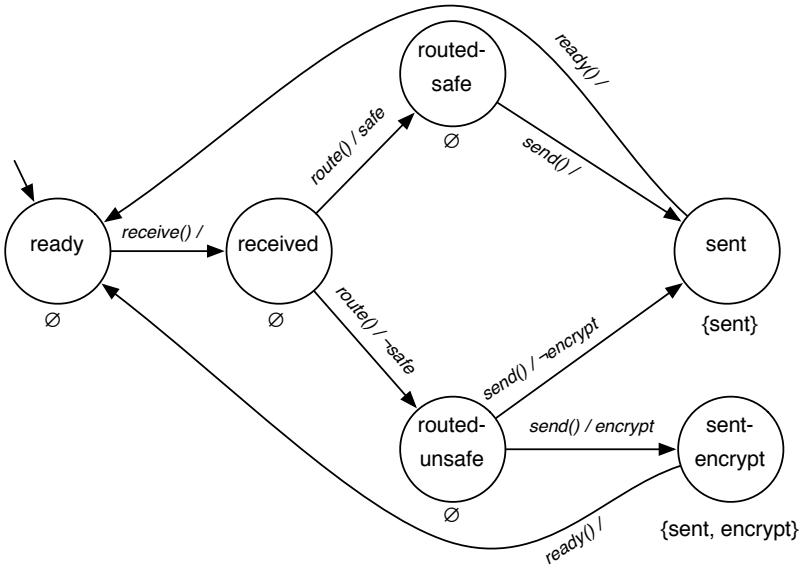


Figure 10.1: The A-FTS modelling the adaptive routing protocol.

that the system is in state *ready*, does not have the feature *encryption* enabled and runs in a $safe$ environment. Then once the action `receive()` is executed, the A-FTS can reach one of the following four macrostates: (`received`, $\emptyset$, $\emptyset$), (`received`, $\emptyset$, $\{safe\}$), (`received`, $\{encryption\}$, $\emptyset$), and (`received`, $\{encryption\}$, $\{safe\}$). The actual macrostate depends on how the environment evolves and how the system decides to reconfigure itself.

Definition 10.3 (Run in A-FTS)
A run in an A-FTS is a sequence of the form

$$(s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \ldots (s_i, c_i, e_i)\alpha_i \ldots$$

where $(s_0, c_0, e_0) \in I \times 2^{F_s} \times 2^{F_e}$ *and where for every* $i \in \mathbb{N}$ *we have* $\gamma(s_i, \alpha_i, s_{i+1})(c_i \cup e_i, c_{i+1} \cup e_{i+1})$.

Note that this definition allows the system and the environment to start in any configuration permitted by the FM. For instance, a run in the A-FTS

described in Example 10.1.1 is

$$(ready, \emptyset, \{safe\})\ \texttt{receive()}\ (ready, \{encrypt\}, \emptyset)\ \texttt{route()}$$
$$(routed\text{-}unsafe, \{encrypt\}, \emptyset)\ \texttt{send()}\ (sent\text{-}encrypt, \{encrypt\}, \emptyset)$$
$$\texttt{ready()}\ (ready, \emptyset, \{safe\}) \ldots$$

Due to the reconfiguration capabilities of A-FTS, the notion of projection is also different from that of FTS. We define the projection of an A-FTS onto a configuration c as the A-FTS obtained by setting its initial configuration to c. The resulting A-FTS is noted $afts_{|c}$ and is such that its set of executions is given by

$$[\![afts_{|c}]\!] = \{(s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \cdots \in ((S \times 2^{F_s} \times 2^{F_e}) \times Act)^\omega$$
$$\mid s_0 \in I \wedge c_0 = c \wedge \forall i \in \mathbb{N} \bullet \gamma(s_i, \alpha_i, s_{i+1})(c_i \cup e_i, c_{i+1} \cup e_{i+1})\}.$$

Then the semantics of an A-FTS is a function that associates a system configuration c with the set of executions where the system starts in configuration c.

Definition 10.4 (A-FTS semantics)
Let $afts$ be an A-FTS. The semantics of $afts$ is the function

$$[\![afts]\!] : 2^{F_s} \to (S \times 2^{F_s} \times 2^{F_e} \times Act)^\omega \bullet [\![afts]\!](c) = [\![afts_{|c}]\!]$$

According to the above semantics, there is a close relation between A-FTS, FTS, and TS. An FTS is an A-FTS where the system and the environment start in any valid configuration and never modifies it, that is

$$\forall (s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \ldots (s_i, c_i, e_i)\alpha_i \cdots \in [\![afts]\!](c_0) \bullet$$
$$\forall i \in N \bullet c_i = c_{i+1} \wedge e_i = e_{i+1}.$$

A sufficient condition for that condition to hold is that $\gamma(s, \alpha, s')(c, c')$ returns $\perp$ whenever $c \neq c'$. In this case, two runs differ only by (1) the executed actions and (2) the states reached by the system. Furthermore, the projection of this FTS onto a configuration c (i.e. an TS) is equivalent to the projection of this special A-FTS to c.

10.2 The AdaCTL Logic

To express properties that a DSPL must satisfy, we use a variant of the fCTL logic [58], which itself extends CTL in the same way as fLTL extends

LTL. Our new logic, called *Adaptive Configuration Time Logic* (AdaCTL), extends the syntax of fCTL to allow further reasoning over the features. Also, its semantics is different because it takes into account that the system and the environment are not always allowed to change their own configuration. In this section, we introduce the syntax and the semantics of AdaCTL and provide an example of property that can be expressed in this logic.

10.2.1 Syntax

We can classify the AdaCTL formulae into three categories. The first type of formula is called *feature* formula. It has the form

$$\Psi ::= [\chi]\Phi$$

where χ is a feature expression and Φ is a *state* formula. To increase readability, when the feature expression χ is equivalent to $\top$, we omit it; that is, for any state formula Φ, we define that

$$\Phi \triangleq [\top]\Phi.$$

A state formula is defined over a set AP of atomic propositions and is built according to the following grammar:

$$\Phi ::= \top \mid a \mid \Psi_1 \wedge \Psi_2 \mid \neg\Psi \mid \mathcal{A}\varphi \mid \mathcal{E}\varphi$$

where $a \in AP$, Ψ, Ψ_1 and Ψ_2 are feature formulae, and φ is a *path formula*. This latter category of AdaCTL formula is defined as follows:

$$\varphi ::= \bigcirc\Psi \mid \Psi_1 \text{ U } \Psi_2 \mid \Psi_1 \text{ R } \Psi_2$$

where Ψ, Ψ_1, and Ψ_2 are feature formulae, $\bigcirc$ is the *next* operator, U is the *until* operator, and R is the *release* operator.

Before providing AdaCTL with a formal semantics, we first explain it intuitively for each type of formula. A feature formula $[\chi]\Phi$ means that if the system is in a given macrostate (s, c, e) such that the configuration of the system and the environment satisfy the feature expression χ (that is, $c \cup e \models \chi$), then s must satisfy the state formula Φ. It is thus very similar to an fCTL formula. The difference is that in fCTL, a feature expression occurs before the top-level state formula only, whereas AdaCTL allows it to occur before any state formula. This provides more flexible ways to reason on the features. In particular, if we want to express that a feature f must be enabled, we may use the formula $[\neg f]\ \bot$, which is satisfied if and only if the system is in a configuration where f is enabled. Moreover, while authorizing feature expressions to occur at any level of a formula would not change the expressiveness of fCTL, it increases that of AdaCTL. For example, since the environment is modelled as a set of features varying over time, AdaCTL

formulae can model changes of objective with respect to what happened in the past.

A state formula is a formula defined over a state. Any macrostate satisfies the formula $\top$. A macrostate (s, c, e) satisfies a if and only if a belongs to the set of atomic propositions in $L(s)$; it satisfies the conjunction of two formulae if and only if it satisfies both. Also, the negation of a formula is satisfied if and only if the formula itself is not satisfied. The AdaCTL operator $\mathcal{E}$ is similar to the existential operator of CTL: a macrostate m satisfies the formula $\mathcal{E}\varphi$ if and only if there exists a path starting from m that satisfies φ. The most subtle difference between AdaCTL and the other two logics lies in the semantics of formulae of the form $\mathcal{A}\varphi$. A macrostate m satisfies $\mathcal{A}\varphi$ if and only if starting from m, the system can ensure by means of reconfigurations that any forthcoming execution will satisfy φ regardless of the environment.

As in CTL, a path π satisfies the AdaCTL path formula $\bigcirc\Psi$ if and only if the first macrostate of π (that is, the macrostate following the initial one) satisfies the feature formula Ψ. A path π satisfies $\Psi_1 \mathrm{U} \Psi_2$ if and only if it eventually reaches a macrostate m_j that satisfies Ψ_2 and every macrostate before m_j on π satisfies Ψ_1. Finally, π satisfies $\Psi_1 \mathrm{R} \Psi_2$ if and only if every macrostate reached by π satisfies Ψ_2 unless a previously reached macrostate satisfied Ψ_1. From the until and the release operator, one can derive two additional, intensively used operators: *eventually* ($\Diamond$) and *forever* ($\Box$). Intuitively, a path π satisfies $\Diamond\Psi$ if and only if there exists a macrostate along π that satisfies Ψ; π satisfies $\Box\Psi$ if and only if every macrostate along this path satisfies Ψ. Formally, these two operators are obtained as follows:

$$
\begin{aligned}
[\chi]\mathcal{E}(\Diamond\Psi) &= [\chi]\mathcal{E}(\top \mathrm{U} \Psi) \\
[\chi]\mathcal{A}(\Diamond\Psi) &= [\chi]\mathcal{A}(\top \mathrm{U} \Psi) \\
[\chi]\mathcal{E}(\Box\Psi) &= [\chi]\mathcal{E}(\top \mathrm{R} \Psi) \\
[\chi]\mathcal{A}(\Box\Psi) &= [\chi]\mathcal{A}(\top \mathrm{R} \Psi)
\end{aligned}
$$

Example 10.2.1 *We now provide an example of AdaCTL formula. Let us consider the A-FTS presented in Example 10.1.1 and the property according which the system must ensure that in an unsafe environment, no packet is sent before it is encrypted. We can express this property as the AdaCTL formula*

$$\mathcal{A}\Box([\neg safe]\mathcal{A}(\neg sent \mathrm{U}[\neg safe]encrypted))$$

The $\Box$ operator is needed because the environment can become unsafe at any moment. According to this formula, from every macrostate where the environment feature $safe$ is disabled, the system must eventually reach a macrostate where either the atomic proposition `encrypted` *is satisfied or the environment is safe again; any macrostate reached in the mean time must be such that the atomic proposition* `sent` *is not satisfied.*

Example 10.2.2 *Assume we model the requirements for a satellite to normally always maintain altitude (a) and make observations at regular intervals (o), but in case the solar panels are damaged (failure d), the second requirement can be dropped. This can be expressed as the formula*

$$\mathcal{A}\Box(\ a \wedge [\neg d]o\ ).$$

In classical CTL, the R operator is derived from the operator of U and the negation. Consequently, $\Box$ is also derived from $\Diamond$ and the negation. However, as we will show further in this section, this definition is not suitable in *AdaCTL* because $\mathcal{A}$ and $\mathcal{E}$ are not dual. Hence the need for considering R as a primitive operator that cannot be derived from the others.

10.2.2 Semantics

Before providing AdaCTL with a formal semantics, we first formalise the notions of strategy for both the system and the environment. Intuitively, a strategy for the system determines how the systems reacts (that is, how it reconfigures itself) according to what happened in the past. We call that a *reconfiguration strategy.*

Definition 10.5 (Reconfiguration strategy)
Let $(S, Act, I, AP, L, d, \gamma)$ be an A-FTS. A reconfiguration strategy is a function

$$Str_C : (S \times 2^{F_s} \times 2^{F_e})^+ \times S \times 2^{F_e} \to 2^{F_s}$$

such that

$$Str_C(s_0, c_0, e_0, \ldots, s_{k-1}, c_{k-1}, e_{k-1}, s_k, e_k) = c_k \Rightarrow (c_k \cup e_k \models fm) \wedge ((c_{k-1} \setminus c_k) \cup (c_k \setminus c_{k-1}) \subseteq F_a).$$

Intuitively, the system modifies its configuration according to the sequence of all the macrostates that have been previously visited, the next state that is reached and the next configuration of the environment. Similarly, we can encode non-deterministic choices and uncontrolled configuration as a strategy for the environment.

Definition 10.6 (Environment strategy)
Let $(S, Act, I, AP, L, d, \gamma)$ be an A-FTS. An environment strategy is a function

$$Str_E : (S \times 2^{F_s} \times 2^{F_e})^+ \to Act \times 2^{F_e}.$$

An environment strategy thus associates the sequence of macrostates that have already been visited with an action and a new configuration for the environment. These definitions of strategy are closely related to those found in the *Alternating Time Logic* (ATL) theory [9]. If the notion of feature were absent, AdaCTL model checking could be regarded as a particular case of ATL model checking. ATL is a logic able to express temporal properties on multi-agent systems and concurrent game structures. It provides a special quantifier $<< A >>$ where A is a set of agents or players working together to meet specific goals. The semantics of this operator makes use of a definition of strategy as well : $<< A >> \varphi$ is satisfied if and only if the players in A can find a strategy such that any execution following this strategy satisfies φ. Given that definition, the AdaCTL quantifier $\mathcal{A}$ is clearly similar to $<< Sys >>$ whereas $\mathcal{E}$ is similar to $<< Sys, Env >>$, Sys being the system and Env being the environment. The difference between the two logics is that in AdaCTL, the transition relation depends on the configuration of both the adaptable and the non-adaptable features. Because of these features, the satisfiability relation is more general than in ATL. Similarly, an A-FTS with a unique initial configuration can be translated into a two-player, turn-based, concurrent game structure. In this game, the players are the system and the environment. The action of the former is the choice of its new configuration. For the latter, it is the choice of an action and of its new configuration. In the end, this work can be regarded as a generalization of turn-based, two-players, concurrent game structures in the same way as FTS generalizes LTS.

Given an initial macrostate $init = (s_0, c_0, e_0)$, an environment strategy Str_E, and a reconfiguration strategy Str_C, applying Str_E and Str_C from $init$ results in a unique run

$$Path(init, Str_C, Str_E) = (s_0, c_0, e_0)\alpha_0(s_1, c_1, e_1)\alpha_1 \dots$$

such that $\forall i \in \mathbb{N}$ we have

$$\begin{aligned}(\alpha_i, e_{i+1}) &= Str_E(s_0, c_0, e_0, \dots, s_i, c_i, e_i)\\ c_{i+1} &= Str_C(s_0, c_0, e_0, \dots, s_i, c_i, e_i, s_{i+1}, e_{i+1}).\end{aligned}$$

This run is *valid* according to $afts$ if and only if it is part of the semantics of $afts$, that is, $Path(Init, Str_C, Str_E) \in [\![afts]\!]$. More generally, we denote by $Path(m, Str_C, Str_E)$ the path starting from a macrostate m induced by the environment strategy $Strat_E$ and the reconfiguration strategy $Strat_C$.

Following the definition of valid run, we define that an environment strategy Str_E is *valid* according to $afts$ if and only if it cannot lead to invalid executions:

$$\forall init \in I \times 2^{F_s} \times 2^{F_e} \bullet \forall Str_C \bullet Path(init, Str_C, Str_E) \in [\![afts]\!].$$

We define similarly the validity of a reconfiguration strategy Str_C:

$$\forall init \in I \times 2^{F_s} \times 2^{F_e} \bullet \forall Str_E \bullet Path(init, Str_C, Str_E) \in [\![afts]\!].$$

From now on, we consider only valid strategies. Then, we define the semantics of AdaCTL as follows.

Definition 10.7 (AdaCTL satisfiability rules)
Let $afts$ be an A-FTS, (s, c, e) one of its macrostates. Then the satisfiability of an AdaCTL feature or state formula by $afts$ in macrostate (s, c, e) is determined according to the following rules:

$$\begin{array}{lcll}
(s,c,e) \models & [\chi]\Phi & \Leftrightarrow c \cup e \not\models \chi \vee (s,c,e) \models \Phi \\
(s,c,e) \models & \top & \Leftrightarrow \top \\
(s,c,e) \models & a & \Leftrightarrow a \in L(s) \\
(s,c,e) \models & \Phi_1 \wedge \Phi_2 & \Leftrightarrow (s,c,e) \models \Phi_1 \wedge (s,c,e) \models \Phi_2 \\
(s,c,e) \models & \neg\Phi & \Leftrightarrow \neg((s,c,e) \models \Phi) \\
(s,c,e) \models & \mathcal{E}\varphi & \Leftrightarrow \exists Str_C \bullet \exists Str_E \bullet Path((s,c,e), Str_C, Str_E) \models \varphi \\
(s,c,e) \models & \mathcal{A}\varphi & \Leftrightarrow \exists Str_C \bullet \forall Str_E \bullet Path((s,c,e), Str_C, Str_E) \models \varphi.
\end{array}$$

The semantics of path formulae is very similar to that of CTL path formulae:

$$\begin{array}{lcll}
\pi \models & \bigcirc\Psi & \Leftrightarrow \pi[1] \models \Psi \\
\pi \models & \Psi_1 \mathrm{U} \Psi_2 & \Leftrightarrow \exists j \geq 0 \bullet \pi[j] \models \Psi_2 \wedge \forall i \leq j \bullet \pi[i] \models \Psi_1 \\
\pi \models & \Psi_1 \mathrm{R} \Psi_2 & \Leftrightarrow (\forall j \geq 0 \bullet \pi[j] \models \Psi_2) \vee \\
 & & \quad (\exists i \bullet \pi[i] \models \Psi_1 \wedge \forall k \leq i \bullet \pi[k] \models \Psi_2)
\end{array}$$

where π is a path, $\pi[0]$ is the initial macrostate of π, and $\pi[i+1]$ is the macrostate following $\pi[i]$ in π.

According to the above, we define that $afts_{|c}$ satisfies an AdaCTL formula Ψ if and only if for any initial state i of $afts$ and environment configuration e, $afts$ satisfies the formula in macrostate (i, c, e); that is,

$$(afts_{|c} \models \Psi) \Leftrightarrow \forall i \in I \bullet \forall e \in 2^{F_e} \bullet (i, c, e) \models \Psi.$$

This leads us to the more general satisfiability relation of an AdaCTL formula by an A-FTS. As for FTS and fCTL (see Chapter 3), this relation is not Boolean. Instead, it is defined as the set of configurations such that when starting in such a configuration, the A-FTS satisfies the formula.

Definition 10.8 (AdaCTL semantics)
Let $afts$ be an A-FTS and Ψ an AdaCTL formula. Then,

$$(afts \models \Psi) = \mathbb{B}(\{c \in 2^{F_s} \mid afts_{|c} \models \Psi\}).$$

Depending on the feature quantifiers and operators, we distinguish between three types of requirements that can be expressed as an AdaCTL formula. A formula Ψ is named an *absolute* requirement if it does not depend on features (i.e. every feature expression occurring in the formula is a tautology). It is named *conditional* if it is non-absolute but only non-adaptable features occur in the feature expressions. It is called *adaptive* if it contains the $\mathcal{A}$ operator. A system is called *adaptive* if and only if it has adaptive requirements and adaptable features.

Definition 10.9 (Resilience)
An adaptive system $afts$ with requirements Φ is called resilient if and only if there is an initial system configuration such that all its requirements are satisfied: $afts \models_F \Phi \neq \bot$.

This implies that, for each adaptive requirement, the system must be equipped with adaptation strategies that allow him to react to any environment (re)configuration, in particular to any failure. For instance, if $afts$ is the A-FTS presented in Example 10.1.1 and Ψ is the adaptive requirement given in Example 10.2.1, we have

$$(afts \models \Psi) = 2^{F_s}$$

since for any initial configuration, there exists a reconfiguration strategy that ensures the satisfaction of Ψ. One such strategy would be to enable the feature *encrypt* as soon as the system reaches the state `received`. However, if setting up this feature requires some time (e.g., for downloading the encryption code), the system has to wait that the feature is enabled before sending the message.

Note that according to the above semantics, $\mathcal{A}$ and $\mathcal{E}$ are not dual. We prove this for the $\bigcirc$ operator.

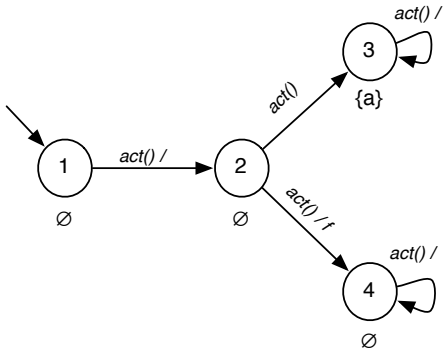


Figure 10.2: Counterexample for the duality laws.

Theorem 10.1
Let Ψ be an AdaCTL feature formula. We have

$$\mathcal{A} \bigcirc \Psi \not\equiv \neg(\mathcal{E} \bigcirc \neg\Psi).$$

PROOF. *Let us assume that $\mathcal{A}$ and $\mathcal{E}$ are dual for the $\bigcirc$ operator. Let us consider the two AdaCTL formulae $\mathcal{A} \bigcirc \mathcal{A} \bigcirc a$ and $\mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a$ where a is an atomic proposition. Then, for any A-FTS $afts$ and system configuration c, if $afts_{|c}$ satisfies the former then it would not satisfy the latter and vice versa. Let $afts$ be the A-FTS shown in Figure 10.2 where f is a feature of the system that can be modified without restriction. It turns out that $afts_{|c} \models \mathcal{A} \bigcirc \mathcal{A} \bigcirc a$ for any configuration c. Indeed, once in state 2 the system can change its configuration such that f is not enabled. In this case, only state 3 is reachable and the formula is thus satisfied regardless of the action chosen by the environment and its configuration. On the other hand, $afts_{|c} \models \mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a$ for any c as well. Once in state 2, if the system change its configuration such that f is enabled, the system can reach state 4. Since $afts$ satisfies both formulae and given that it is impossible for an A-FTS to satisfy both a formula and its negation, the duality law does not hold.* □

The above counterexample also denies the duality law for the $\Diamond$, the $\Box$, the U and the R operators. Since the proof is very similar, we omit it.

Theorem 10.2
Let Ψ be an AdaCTL feature formula. We have

$$\begin{aligned}\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2) &\neq \neg(\mathcal{E}(\neg\Psi_1 \mathrm{R} \neg\Psi_2))\\ \mathcal{E}(\Psi_1 \mathrm{U} \Psi_2) &\neq \neg(\mathcal{A}(\neg\Psi_1 \mathrm{R} \neg\Psi_2))\\ \mathcal{A}\Diamond\Psi &\neq \neg(\mathcal{E}\Box\neg\Psi)\\ \mathcal{E}\Diamond\Psi &\neq \neg(\mathcal{A}\Box\neg\Psi).\end{aligned}$$

This implies that $\mathcal{A}$ cannot be directly expressed in terms of $\mathcal{E}$. Thus, when model checking an A-FTS against an AdaCTL formula, we have to consider the operator $\mathcal{A}$ and $\mathcal{E}$ separately. On the contrary, the usual *expansion laws* still hold. Basically, an expansion law allows one to express an operator in terms of formulae that the current state and the next state must satisfy to satisfy the formula defined by the operator. For instance, the expansion law for $\mathcal{A}(\Psi_1 \mathrm{U} \Psi 2)$ expresses that this formula is satisfied if and only if Ψ_2 immediately holds or if Ψ_1 holds and the formula holds in the next state. As for CTL, the AdaCTL model-checking algorithms make intensive use of these laws, as we will see in the next section.

Theorem 10.3
Let Ψ, Ψ_1, Ψ_2 be AdaCTL formulae. Then we have the following expansion laws:

$$\begin{aligned}\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2) &\equiv \Psi_2 \vee (\Psi_1 \wedge \mathcal{A} \bigcirc \mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))\\ \mathcal{A}(\Psi_1 \mathrm{R} \Psi_2) &\equiv \Psi_2 \wedge (\Psi_1 \vee \mathcal{A} \bigcirc \mathcal{A}(\Psi_1 \mathrm{R} \Psi_2))\\ \mathcal{A}(\Diamond\Psi) &\equiv \Psi \vee \mathcal{A} \bigcirc \mathcal{A}\Diamond\Psi\\ \mathcal{A}(\Box\Psi) &\equiv \Psi \wedge \mathcal{A} \bigcirc \mathcal{A}\Box\Psi\\ \mathcal{E}(\Psi_1 \mathrm{U} \Psi_2) &\equiv \Psi_2 \vee (\Psi_1 \wedge \mathcal{E} \bigcirc \mathcal{E}(\Psi_1 \mathrm{U} \Psi_2))\\ \mathcal{E}(\Psi_1 \mathrm{R} \Psi_2) &\equiv \Psi_2 \wedge (\Psi_1 \vee \mathcal{E} \bigcirc \mathcal{E}(\Psi_1 \mathrm{R} \Psi_2))\\ \mathcal{E}(\Diamond\Psi) &\equiv \Psi \wedge \mathcal{E} \bigcirc \mathcal{E}\Diamond\Psi\\ \mathcal{E}(\Box\Psi) &\equiv \Psi \wedge \mathcal{E} \bigcirc \mathcal{E}\Box\Psi.\end{aligned}$$

PROOF. *We prove only the expansion law of $\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)$. The other proofs involving the $\mathcal{A}$ operator either can be derived from this one or follow a similar pattern. The proofs related to $\mathcal{E}$ are implied from the expansion laws in CTL. For any $m = (s, c, e)$, Str_E, and Str_C, the initial state of $Path(m, Str_C, Str_E)$ is m and thus depends on neither Str_C nor Str_E. Ac-*

cordingly, the semantics of $\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)$ *is equivalent to*

$$(m \models \Psi_2) \vee ((m \models \Psi_1) \wedge \exists Str_C \bullet \forall Str_E \bullet \\ Path(Path(m, Str_C, Str_E)[1], Str_C, Str_E) \models \Psi_1 \mathrm{U} \Psi_2 \quad (10.1)$$

On the other hand,

$$m \models (\Psi_2 \vee (\Psi_1 \wedge \mathcal{A} \bigcirc \mathcal{A}(\Psi_1 \mathrm{U} \Psi 2)))$$

is equivalent to

$$(m \models \Psi_2) \vee ((m \models \Psi_1) \wedge \exists Str'_C \bullet \forall Str'_E \bullet \\ Path(m, Str'_C, Str'_E) \models \bigcirc \mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)$$

which can be re-written as

$$(m \models \Psi_2) \vee ((m \models \Psi_1) \wedge \exists Str'_C \bullet \forall Str'_E \bullet \exists Str''_C \bullet \forall Str''_E \bullet \\ Path(Path(m, Str'_C, Str'_E)[1], Str''_C, Str''_E) \models \Psi_1 \mathrm{U} \Psi_2 \quad (10.2)$$

We immediately have that Equation (10.1) implies Equation (10.2); in this case, we have $Str_C = Str'_C = Str''_C$*. Next, we define the strategy* Str'''_C *such that* Str'''_C *behaves like* Str'_C *for the first transition, and like* Str'''_C *for the subsequent ones. Then for any* Str'''_E *we have that*

$$Path(Path(m, Str'''_C, Str'''_E)[1], Str'''_C, Str'''_E) \models \Psi_1 \mathrm{U} \Psi_2.$$

Equations (10.1) and (10.2) are thus equivalent and we have proven the expansion law. □

The expansion laws for $\mathcal{A}(\varphi)$ imply that if Str_C is the reconfiguration strategy such that $\forall Str_E \bullet Path(m, Str_C, Str_E) \models \varphi$ then every macrostate m' reached on this path is such that $m' \models \mathcal{A}(\varphi)$.

10.3 Algorithms

As previously mentioned, model checking an A-FTS against an AdaCTL formula comes down to identifying the initial configurations such that for any initial state and initial environment configuration, the A-FTS satisfies the formula when the system starts in such a configuration. In this section, we propose an algorithm to compute the satisfaction relation between an A-FTS $afts$ and an AdaCTL formula Ψ. For the sake of readability and conciseness, we present only the algorithms in their explicit form. Following Classen *et al.*'s work on fCTL model checking [58], we can transform them into symbolic algorithms by using the same encoding techniques. The basic idea is to encode states, configurations, and the transition relation as

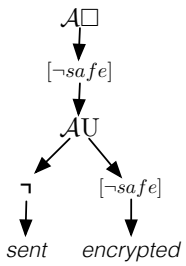


Figure 10.3: Parse tree of $\mathcal{A}(\Box[\neg safe]\mathcal{A}(\neg sent\ \mathrm{U}\ [\neg safe]encrypted))$.

Boolean functions. Then, symbolic algorithms work with those functions instead of their explicit counterpart (see [58] for more details). Note that the conversion of the transition relation in A-FTS is facilitated since it is already defined as a Boolean function.

10.3.1 AdaCTL Model Checking

We first decompose Ψ into its *parse tree*. Basically, the parse tree of a formula is a tree such that each node represents a subformula of Ψ, the root is Ψ itself, and the leaves are atomic propositions. For example, the parse tree of the AdaCTL formula given in Example 10.2.1 is shown in Figure 10.3. Then, starting from the leaves and for every subformula Ψ', we compute the set of macrostates

$$Sat(\Psi') = \{(s, c, e) \in S \times 2^{F_s} \times 2^{F_e} \mid (s, c, e) \models \Psi'\} \tag{10.3}$$

for every subformula Ψ' of Ψ. Once we have determined the set of macrostates satisfying the whole formula Ψ, we infer the set $(afts \models \Psi)$. This method allows us to decompose the verification of a formula into smaller and independent problems. It is thus similar to the fCTL and the standard CTL model-checking algorithms [58, 46]. However, it has to cope with (1) macrostates instead of states, (2) dynamic features, and (3) the $\mathcal{A}$ quantifier, which does not exist in the other two logics.

The computation of the set of macrostates satisfying feature and state formulae directly follows from their semantics.

$$\begin{aligned} Sat([\chi]\Phi) &= \{(s, c, e) \in M \mid c \cup e \not\models \chi\} \cup Sat(\Phi) \\ Sat(\top) &= M \\ Sat(a) &= \{(s, c, e) \in M \mid a \in L(s)\} \\ Sat(\Psi_1 \wedge \Psi_2) &= Sat(\Psi_1) \cap Sat(\Psi_2) \\ Sat(\neg\Psi) &= M \setminus Sat(\Psi) \end{aligned}$$

where M is the set of macrostates.

The first step towards computing a set of the form $Sat(\mathcal{E}\varphi)$ or $Sat(\mathcal{A}\varphi)$ is the definition and the computation of predecessors set. The notion of predecessors is, however, different for the $\mathcal{E}$ quantifier and the $\mathcal{A}$ quantifier.

In the former case, a macrostate m is an $\mathcal{E}$-predecessor of m' if and only if from m, the macrostate m' can be reached in one transition. More generally, let $\mathcal{S}$ be a set of macrostates. Then the $\mathcal{E}$-predecessors set of $\mathcal{S}$, noted $Pre_{\mathcal{E}}(\mathcal{S})$, is defined as the set of macrostates from which a macrostate in $\mathcal{S}$ can be reached in one transition. Formally,

$$Pre_{\mathcal{E}}(\mathcal{S}) = \{(s,c,e) \mid \exists \alpha \in Act, (s',c',e') \in \mathcal{S} \bullet \gamma(s,\alpha,s')(c \cup e, c' \cup e')\}.$$

In the latter case, we define that m is an $\mathcal{A}$-predecessor of m' if the system can come up with a valid strategy such that when in m, it is ensured that m' will be reached after the execution of the next transition. Similarly to the previous case, we define the $\mathcal{A}$-predecessor set of a set of macrostates $\mathcal{S}$. Intuitively, it is the set of macrostates such that the system can ensure that after the execution of the next transition, it will reach a macrostate of $\mathcal{S}$. Given that the system chooses its next configuration after the next state and the new environment configuration have been determined, it is defined as

$$\begin{aligned} Pre_{\mathcal{A}}(\mathcal{S}) = \{(s,c,e) \mid \forall \alpha \in Act, s' \in S, e' \in 2^{F_e} \bullet \\ (\exists c'' \in 2^{F_s} \bullet \gamma(s,\alpha,s')(c \cup e, c'' \cup e') \Rightarrow \\ (\exists c' \in 2^{F_s} \bullet \gamma(s,\alpha,s')(c \cup e, c' \cup e') \wedge (s',c',e') \in \mathcal{S})). \end{aligned}$$

Let

$$D(s,c,e,\alpha,s',e') = \{(s',c',e') \in S \times 2^{F_s} \times 2^{F_e} \mid \gamma(s,\alpha,s')(c \cup e, c' \cup e')\}$$

then $Pre_{\mathcal{A}}(\mathcal{S})$ is equivalent to

$$\bigcap_{\alpha,s',e'} \{(s,c,e) \mid (D(s,c,e,\alpha,s',e') = \emptyset) \vee (D(s,c,e,\alpha,s',e') \cap \mathcal{S} \neq \emptyset)\}.$$

Since the semantics of $\mathcal{E}$ is similar to that of the existential quantifier in CTL, the set of macrostates satisfying a formula of the form $\mathcal{E}\varphi$ is computed as in the basic CTL model-checking algorithm. $Sat(\mathcal{E} \bigcirc \Psi)$ is the set of macrostate such that in one transition, the system can reach a state s', be in configuration c' and the environment can be in a configuration e' such that (s',c',e') satisfies φ. Formally, we have

$$Sat(\mathcal{E}(\bigcirc\Psi)) = Pre_{\mathcal{E}}(Sat(\Psi)).$$

On the other hand, $\mathcal{E}(\Psi_1 \mathrm{U} \Psi_2)$ is the smallest set $\mathcal{S}$ satisfying

$$Sat(\Psi_2) \subseteq \mathcal{S}$$
$$Sat(\Psi_1) \cap Pre_{\mathcal{E}}(S) \subseteq \mathcal{S};$$

$\mathcal{E}(\Psi_1 \mathrm{R} \Psi_2)$ is the largest set $\mathcal{S}$ satisfying

$$\mathcal{S} \subseteq Sat(\Psi_2)$$
$$\mathcal{S} \subseteq Sat(\Psi_1) \cup Pre_{\mathcal{E}}(\mathcal{S}).$$

Both can be computed through fixed-point computation [46]. The corresponding algorithms being identical to their CTL counterpart, we omit them here.

We focus now on the $\mathcal{A}$ quantifier. Basically, the satisfaction sets have a similar definition than those for the $\mathcal{E}$ quantifier; the difference is that the $\mathcal{A}$ quantifier requires the computation of the $\mathcal{A}$-predecessors instead of the $\mathcal{E}$-predecessors. For the next operator, we have the following.

Theorem 10.4

$$Sat(\mathcal{A} \bigcirc \Psi) = Pre_{\mathcal{A}}(\Psi)$$

PROOF. *Follows from the semantics of* $\mathcal{A} \bigcirc \Psi$ *and the definition of* $Pre_{\mathcal{A}}(\Psi)$. □

We obtain $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$ through a fixed-point computation. This result relies on the following theorem.

Theorem 10.5
$Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$ *is the smallest set* $\mathcal{S}$ *satisfying*

$$Sat(\Psi_2) \subseteq \mathcal{S} \tag{10.4}$$
$$Sat(\Psi_1) \cap Pre_{\mathcal{A}}(\mathcal{S}) \subseteq \mathcal{S} \tag{10.5}$$

PROOF. *First, it directly follows from the expansion law of the* U *operator (see Theorem 10.3) that* $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$ *satisfies Equations (10.4–10.5). It remains to show that any set* $\mathcal{S}$ *satisfying those Equations is a superset of* $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$.

Let $m \in Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$. *We distinguish between the case where* $m \in Sat(\Phi_2)$ *and the case where* $m \notin Sat(\Phi_2)$.

(a) *If* $m \in Sat(\Phi_2)$ *then* $m \in \mathcal{S}$ *by Equation 10.4.*

(b) *Otherwise, by definition of* $Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2))$*, we have*

$$\exists Str_C \bullet \forall Str_E \bullet Path(m, Str_C, Str_E) \models \Psi_1 \mathrm{U} \Psi_2.$$

It means that Str_C *ensures that from* m*, we eventually reach a macrostate that is in* $Sat(\Psi_2)$*. Let* k *be the largest number of transitions needed by* Str_C *to reach from* m *such a macrostate, and let* m_k *this macrostate. Then* $m_k \in \mathcal{S}$ *by Equation 10.4. Before reaching* m_k*, we must first reach a macrostate* $m_{k-1} \in Sat(\Psi_1)$ *such that from* m_{k-1}*,* Str_C *ensures to reach* m_k *in one transition regardless of the strategy of the environment. Such a macrostate is reached in at most* $k-1$ *transitions, and belongs to* $Pre_{\mathcal{A}}(\{m_k\}) \subseteq Pre_{\mathcal{A}}(Sat(\Psi_2))$*, and is thus in* $\mathcal{S}$ *by Equation (10.5). By induction on the maximum number of transitions needed to reach a macrostate in* $Sat(\Psi_2)$*, we obtain that* $m \in \mathcal{S}$*.*

The proof is then complete. □

In order to compute $Sat(\Psi_1 \mathrm{U} \Psi_2)$, we define the function

$$T_{\mathrm{U}} : 2^M \to 2^M \bullet T_{\mathrm{U}}(\mathcal{S}) = \mathcal{S} \cup (Pre_{\mathcal{A}}(\mathcal{S}) \cap Sat(\Phi_1)).$$

Then, according to the Knaster-Tarski theorem and following Theorem 10.5, this set is the fixed-point of the function T_{U} when it is first applied to $Sat(\Psi_2)$, that is,

$$Sat(\mathcal{A}(\Psi_1 \mathrm{U} \Psi_2)) = \mathcal{S}_i \bullet \forall j \geq i \bullet \mathcal{S}_j = \mathcal{S}_i$$

where $\forall j \in \mathbb{N}$

$$\mathcal{S}_0 = Sat(\Psi_2)$$
$$\mathcal{S}_{j+1} = T(\mathcal{S}_j).$$

The computation of $Sat(\mathcal{A}(\Psi_1 \mathrm{R} \Psi_2))$ follows a very similar procedure. For this reason, we omit the proof.

Theorem 10.6
$Sat(\mathcal{A}(\Psi_1 \mathrm{R} \Psi_2))$ *is the largest set* $\mathcal{S}$ *satisfying*

$$\mathcal{S} \subseteq Sat(\Psi_2)$$
$$\mathcal{S} \subseteq Sat(\Psi_1) \cup Pre_{\mathcal{A}}(\mathcal{S}).$$

Accordingly, this set can be computed as the fixed-point of the function

$$T_{\mathrm{R}} \quad : \quad 2^{(S\times 2^{F_s}\times 2^{F_e})} \quad \to \quad 2^{(S\times 2^{F_s}\times 2^{F_e})} \bullet T_{\mathrm{R}}(\mathcal{S}) \quad = \quad \mathcal{S} \cap Pre_{\mathcal{A}}(\mathcal{S}).$$

first applied to $Sat(\Psi_2)$.

Example 10.3.1 *We illustrate the definitions of $\mathcal{E}$-predecessors and $\mathcal{A}$-predecessors, as well as the above model-checking algorithm. Let us consider the small A-FTS shown in Figure 10.2, which we verify against the formulae $\mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a$ and $\mathcal{A} \bigcirc \mathcal{A} \bigcirc a$. First, the algorithm determines which macrostates satisfy the atomic proposition a:*

$$Sat(a) = \{(3, c, e)\}.$$

Since $\neg a$ occurs in the first formula, the algorithm computes the corresponding satisfaction set by complementing the above:

$$Sat(\neg a) = \{(i, c, e) \mid i \in \{1, 2, 4\}\}.$$

We focus on $\mathcal{E}$-predecessors first. A macrostate satisfies $\mathcal{E} \bigcirc \neg a$ if and only if from this macrostate, the system may reach a set in $Sat(\neg a)$ in one transition. Hence,

$$Sat(\mathcal{E} \bigcirc \neg a) = \{(i, c, e) \mid i \in \{1, 4\} \vee (i = 2 \wedge f \in c)\}.$$

Indeed, from state 1 the system can reach state 2 regardless of its configuration and that of the environment; from state 2 it can reach state 4 if feature f is enabled; and it can always loop on state 4. For the same reasons, the set of macrostates satisfying the whole property is given by

$$Sat(\mathcal{E} \bigcirc \mathcal{E} \bigcirc \neg a) = \{(i, c, e) \mid i \in \{1, 4\} \vee (i = 2 \wedge f \in c)\}.$$

A macrostate satisfies $\mathcal{A} \bigcirc a$ if and only if from this macrostate, the system can ensure it will reach a macrostate satisfying a. Regardless of the configuration of the system and the environment, the former will remain in state 3 forever from the moment it reaches this state. Moreover, if the system is in state 2 and feature f is disabled, it will necessarily reach state 3 after the next transition. From state 1, the system cannot reach state 3 in one transition. The corresponding satisfaction set is thus:

$$Sat(\mathcal{A} \bigcirc a) = \{(3, c, e)\} \cup \{(2, c, e) \mid f \notin c\}$$

The definition of $Sat(\mathcal{A} \bigcirc \mathcal{A} \bigcirc a)$ is identical except that this time, the system can satisfy the formula from state 1. The configuration of the system does not matter here since the system may modify it once it reaches state 2. We have

$$Sat(\mathcal{A} \bigcirc \mathcal{A} \bigcirc a) = \{(3, c, e)\} \cup \{(2, c, e) \mid f \notin c\} \cup \{1, c, e\}.$$

10.3.2 Time Complexity

We now discuss the computational time complexity of checking an A-FTS $afts$ against an arbitrary AdaCTL formula Ψ. Our algorithm recursively computes the satisfactions sets of the subformulae of Ψ. Its time complexity is thus linear in the size of Ψ. The satisfaction sets that are the most costly to compute are those for the U and the R operators. Let us assume that we encode the satisfaction sets and the transition relation symbolically. As in classical CTL model checking, the number of iterations to compute $Sat(\mathcal{E}(\Phi_1 \mathrm{U} \Phi_2))$ and $Sat(\mathcal{E}(\Phi_2 \mathrm{R} \Phi_2))$ is bounded by the number of macrostates, i.e., $|S|.2^{|F_s|}.2^{|F_e|}$. This is because when computing the corresponding smallest (resp. greatest) fixed-point, if the fixed-point has not been reached then the application of the corresponding function removes (resp. adds) at least one element. Computing the sets $Sat(\mathcal{A}(\Phi_1 \mathrm{U} \Phi_2))$ and $Sat(\mathcal{A}(\Psi_1 \mathrm{R} \Phi_2))$ is more costly because each application of the functions T_{U} and T_{R} requires the computation of $Pre_{\mathcal{A}}(\mathcal{S})$. The time complexity of this computation is bounded by the number of states multiplied the number of environment configurations, that is, $|S|.2^{|F_e|}$. Since the number of needed applications of T_{U} and T_{R} is also bounded by the number of macrostates, we obtain the following result.

Theorem 10.7
The time complexity of model checking an A-FTS against an AdaCTL formula Ψ is bounded by $\mathcal{O}(|S|^2.2^{2.|F_e|}.2^{|F_s|})$.

Although it is theoretically dominated by $2^{(2.|F_e|)}$ and is thus in EXPTIME, in practice $|S|^2$ is often bigger.

10.4 Perspectives

We have presented a well-founded framework for the modelling and analysis of (self-)adaptive systems. We proposed a fundamental model, A-FTS, and a logic, AdaCTL, that are the basis for algorithms for analyzing resilience. This brings a number of benefits:

1. A sound theoretical basis;
2. An integration of static adaptation and dynamic adaptation, in its two variants: external adaptation and self-adaptation by applying a pre-programmed change at the adequate point of time;
3. A clear, checkable definition of resilience;

4. Providing counterexamples when resilience fails.

These benefits mainly impact the predictability of self-adaptive systems.

However, we are well aware that many topics need further development before our methodology can be used routinely by engineers. A-FTS are just a fundamental model, that is used by the tools but leads to lengthy descriptions, difficult to manage by humans. It is therefore important to provide more manageable *high-level languages*, that can be compiled (on-the-fly) to A-FTS. These languages need to cover different *levels of abstraction*: the most abstract levels allowing a rapid analysis, while the most detailed can be directly used to generate executable code. This raises the question of *refinement*: can we guarantee that the analysis performed at the abstract level remains valid at the more concrete level? This question can be solved e.g. using alternating refinement [10]. This refinement should be *compositional*, so that modules of the system can be detailed independently, allowing teams to operate in parallel, on one hand, and analysis tools to cope with complexity, on the other.

Some features have a cross-cutting nature: we cannot simply add a module to the system to realize them. That is why we designed A-FTS with a low grain, where individual transitions can refer to features. As a consequence, first, features are spread over the whole A-FTS, and may be difficult to grasp; second, the addition of a supplementary feature is difficult, since each transition might need a revision. A possibility is to use (extensions of) the *aspect-oriented* approach to maintain a more localized and independent description of each feature (previous work on this topic includes [58, 65, 92, 133]). The addition of a new feature will then hopefully be more understandable. We consider as still unrealistic, though, to hope that all features can be developed without being aware of other features, and that the weaving process will solve all emergent interactions. This raises the problem of defining, detecting and helping to solve *feature interactions*. A number of interesting approaches [101, 34] are available, but the problem is still pressing and largely unsolved.

A-FTS is based on the notion of a global state, so that all the information is available to both the system and its environment, and the strategies computed can rely on the unbounded past. In our setting, it is demonstrated that a bounded amount of information about the past is sufficient (finite memory). Techniques such as antichains [89] can be used to heuristically compute strategies that require a small memory. However, assuming complete observation is not realistic in most systems, and in general we need to switch to the notion of *partial observation* [43, 14]. Unfortunately, the problem becomes highly complex and often even undecidable [37]. The known (expensive) techniques rely on building all possible evolutions of the environment corresponding to the observations so far [43, 118]. On the other hand, we have assumed that the self-adaptive system can choose among a set

of preprogrammed dynamic system features. This is more powerful than it seems, since the system can, if needed, chain several features to construct a complex plan to resist to a hostile environment. In particular, we can model *self-programming* systems by giving as features, the atomic instructions to be assembled. If the atomic instructions are infinite in number, however, our technique does not apply. In the long term, this might become a relevant challenge for deeply self-adaptive autonomous systems.

We modelled failures as a subtype of environment features, which allows us to describe failure modes and effects. To have a realistic analysis of failures, we need to also model their *probabilities*, and to integrate (at least) classical methods for failure and reliability analysis [84]. Also, many systems operate in a continuous physical environment. *Hybrid models* can be used to integrate the modelling of the continuous and discrete parts [162, 73]. Furthermore, a realistic strategy for a hybrid system must be robust, i.e. able to cope with small disturbances [137]. A first step done in this direction is to incorporate *real-time* [163, 138, 65, 66].

In summary, we hope to complement our approach with others, as needed by the vast and multi-disciplinary area of self-adaptive systems, that are expected to progressively enter all domains [95], starting with areas where the need is more pressing, among which controllers for space [41], networked systems [102], robotics [169], anti-intrusion systems [151], and cloud computing [81]. Another interesting perspective is to combine our work with that of Filieri *et al.* [88], as it could allow us to quantify the impact of adding or removing features at runtime in terms of reliability, performance or even energy consumption.

Chapter 11

Synthesis of Product-Line Controllers

As any model checking approach, all the developments presented so far make the assumption that the verified model is a precise specification of the system's behaviour. Capturing a precise specification of an SPL is, however, a not-yet-addressed requirements engineering challenge. Many product variants must be specified, and complex behaviour may arise due to subtle interactions between features. Consequently, the requirements have to be carefully revised to ensure that the features can be combined consistently. If inconsistencies remain undetected, they may persist in many products while the cost of correcting them becomes increasingly higher. To cope with this complexity, it is important that engineers have intuitive but precise methods to specify these systems, as well as tools to automatically find inconsistencies. With regard to these criteria, scenario-based modelling languages appear as an adequate means for specifying system behaviour. If the specification is consistent, it is possible to synthesize a controller describing how the system must behave in order to satisfy the specification. Only a couple of attempts were made to tackle the synthesis of SPL controllers [99, 100]. The first approach [99] is complete but not sound (that is, it may report false errors) and is limited to consistency checking. The second [100] actually provides a sound and complete product-line synthesis approach, but it relies on a procedure that synthesizes a product incrementally based on a previously synthesized one. Its exploitation of the commonalities between products is thus limited to pairs of products.

In this chapter, we propose an alternative method that extends the theory of controller synthesis to product lines. To that aim, we lean on the developments realised in FTS-based model checking. We first provide a formal definition of the product-line synthesis problem. Then we propose efficient algorithms to synthesize a controller for each product while maximizing reuse. The single-system synthesis problem comes down to reach-

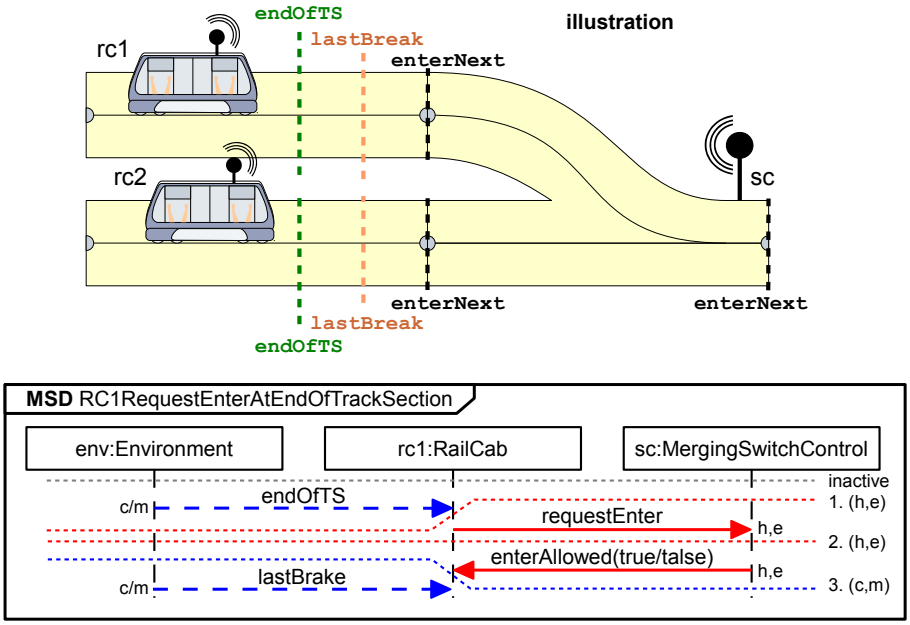


Figure 11.1: An MSD specifying a switch controller.

ability game solving [110, 35]. Since we deal with SPLs, this reachability game must include constructs to represent variability, thereby allowing the synthesis of many controllers from similar specifications. As a preliminary step, we thus define variability-aware reachability game. Then we design algorithms to solve these novel reachability games, which make use of optimisation principles similar to those developed for FTS.

11.1 The Product-Line Synthesis Problem

An essential ingredient of synthesis is a specification of what is a correct behaviour of the considered system. An intuitive way to formulate this specification is to define it as a set of scenarios. Concretely, we assume this specification to be given as a set of sequence diagrams. We more precisely consider Modal Sequence Diagrams (MSDs), a modal variant of sequence diagrams [111], and more particularly of live sequence charts [28]. MSDs allows one to specify interaction between objects that are either part of the system (and thus whose behaviour is controlled by the system) or outside this system (that is, part of an uncontrolled environment). They thus compose the specification of open, reactive systems. An example of MSD is given in Figure 11.1. It is composed of three objects `env`, `rc1`, and `sc`, each of which has its own lifeline. Objects can exchange messages, whose order depends on the position on the lifeline of the objects: the higher a message lies, the sooner it is sent. The life of the system can thus be regarded as a set of

cuts, i.e. time intervals between two occurrences of message. A message is enabled if the current cut lies right before this message. Any message has an execution kind (monitored/dashed or executed/continuous) and a temperature (cold/blue or hot/red). On the one hand, the execution kind determines whether the message *may* occur (i.e. be monitored) or *must* occur (i.e. be executed); this is analogous to transitions in modal transition systems (see Section 4.1). On the other hand, the temperature of a message m determines whether other messages may occur while m is expected. If m is hot then the occurrence of another message of the MSD results in a *safety violation* of the specification. If m is cold and another message of the MSD occurs, the scenario aborts but the specification is still satisfied; this is named a *cold violation*. In addition to temperature and execution kind, we also distinguish between environment and system messages. Basically, the system can control the sending of its own messages, but not of those of the environment. MSDs provide other constructs such as environment assumptions, parametrized messages, forbidden messages, etc. We do not give more details here since these are very specific, whereas our synthesis algorithms are not dependent of the used specification language.

One scenario is generally not enough to specify a system; an MSD specification is rather composed of a set of MSDs evolving in parallel. The current cuts in these MSDs altogether determine which messages may or must occur next, that is, according to the enabled messages and their characteristics. When a message occurs, every MSD in which it is active moves to its next cut. If the message is the first in a given MSD, this MSD becomes *active* as it moves to its second cut. It remains active until all its messages have occurred, namely until it reaches its final cut or aborts due to a cold violation. We assume that only one copy of a given MSD can be active at a time. Since multiple MSDs can be active at the same time, it may happen that their respective specification cannot be both fulfilled. Harel and Marelly formalised the concept of consistency via the definition of the so-called *play-out* semantics [112]. Basically, this semantics specifies that the system and the environment send messages that can activate MSDs or change the current cut of active MSDs, such that:

1. both the system and the environment send only enabled messages;
2. the system can send an infinite number of messages between two successive messages of the environment.

A consistent execution of the play-out semantics is an execution that does not result in a safety violation of the specification, and where every active MSD eventually becomes inactive. The undesired case where the system never reaches such a state is named *liveness violation*. Then an MSD specification is consistent if and only if, given any sequence of environment messages, the system can find a sequence of system messages that leads to a

consistent execution. Similarly to adaptive systems (see Chapter 10), we define a system (resp. environment) strategy as the sequence of system messages (resp. environment messages) chosen by the system (resp. the environment). Accordingly, we define the synthesis problem as follows.

Definition 11.1 (Synthesis problem)
Given an MSD specification, the MSD synthesis problem consists in finding a system strategy that leads to a consistent execution regardless of the environment strategy.

We also name *controller* a strategy yielded by the synthesis process. The synthesis problem thus requires to determine if an MSD specification is consistent and, in this case, to return a controller for the system.

Example 11.1.1 *Figure 11.2 shows an example of inconsistent MSD specification. After the messages* `enterAllowed(true)` *and* `requestEntered` *occur, the MSD* `RC1CoordinateSwitchEntry` *specifies that a given sequence of messages must occur, including a message* `enterAllowed(true)` *from* `sc` *to* `rc1` *and forbids the message* `unregister` *from* `rc1` *to* `sc` *to occur until the MSD reaches its final cut. On the contrary, the MSD* `RC1EnterDisallowedWhenSwitchBlocked` *forbids this message to occur until* `rc2` *sends a message* `unregister` *to* `sc`*.*

In the case of SPLs, different products can have different specification scenarios. Since variability between software products are commonly captured into features, we propose to combine the above specification formalism with feature-based variability modelling.

Definition 11.2 (Scenario-based specification of SPL)
Let M be a set of MSDs and fm be an FM over a set F of features. A scenario-based specification is a total function $msdf : M \to 2^{2^F}$ that associates every MSD with a feature expression encoding the products in which the MSD is executed. The MSD specification of a product p is the subset of M whose associated feature expression is satisfied by the product, that is, a set $M' \subseteq M$ such that $m \in M' \Leftrightarrow p \models msdf(m)$.

When two products have different MSD specifications, the synthesis can produce two different controllers for them. As for model checking, state-of-the-art synthesis algorithms can thus be applied only in a product-by-product manner, i.e. once per individual valid product. As an alternative,

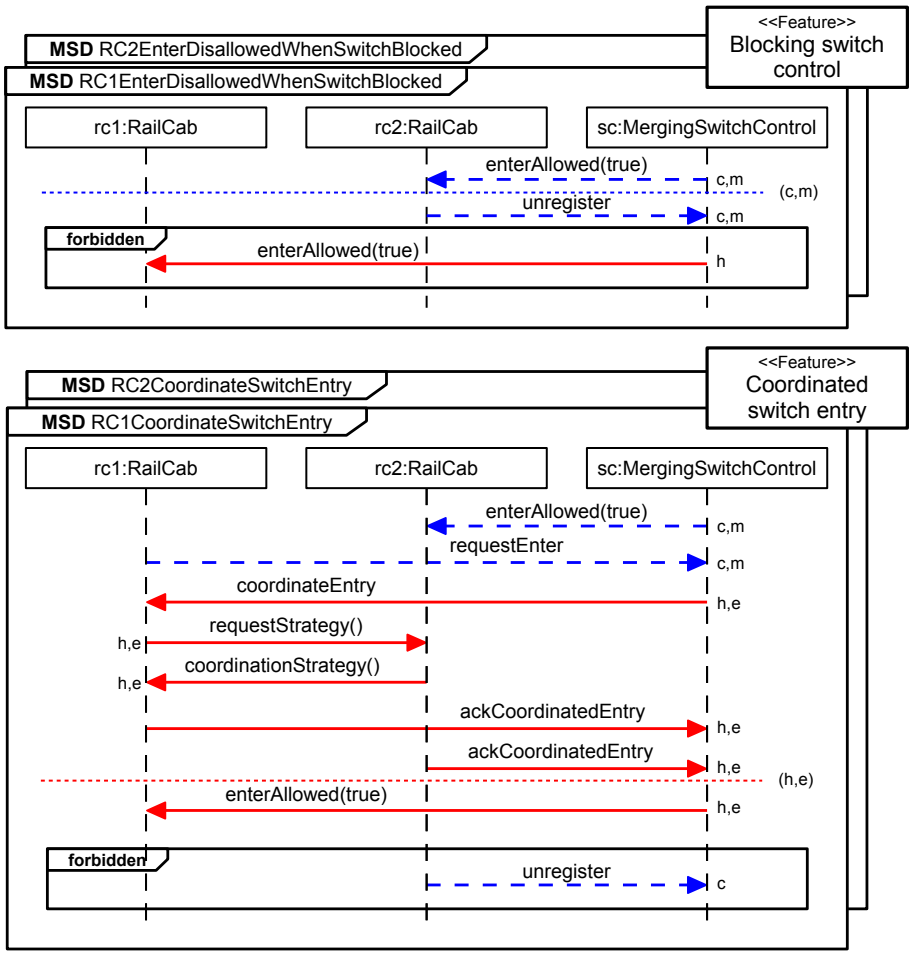


Figure 11.2: An inconsistent MSD specification.

we propose to extend these algorithms with new heuristics inspired by the FTS theory.

A particular form of our SPL specification language consists in associating every MSD with a single feature [99, 100]. In this case, it follows the *compositional* paradigm (see Section 1.3), that is, features are specified independently from each other. A limitation of this approach is that one cannot represent the combined behaviour of features, e.g. to solve feature interactions. This is the reason why we extended the specification language with the capability to associate MSDs to feature expressions instead of features.

11.2 Featured Reachability Games

Like the verification of adaptive systems (see Chapter 10), the synthesis process can be regarded as a game between the system and its environment.

The actions they can execute at a given point of time are their respective enabled messages. The winning condition for the system is that it must reach infinitely often a state where all MSDs are inactive (henceforth called an accepting state) while avoiding safety violations (represented as a *failure state*). Given the subset of MSD constructs we consider, the accepting state always exists and is unique. Other constructs like environment assumptions would yield additional accepting states. Even though we do not consider these constructs, for the sake of generality our approach supports multiple accepting states. In what follows, we formalise controller synthesis from an MSD specification via the definition of Büchi reachability game [110].

Definition 11.3 (Büchi Reachability Game)
A Büchi Reachability Game (BRG) is a tuple $brg = (Q, \Sigma, \delta_c,, \delta_u, q_0, A, B)$ *where*

- Q *is a set of states;*
- Σ *is the alphabet;*
- $\delta_c \subseteq Q \times \Sigma \times Q$, *is the set of controllable transitions;*
- $\delta_u \subseteq Q \times \Sigma \times Q$, *is the set of uncontrollable transitions, with* $\delta_c \cap \delta_u = \emptyset$ *and*

  $$\forall q \bullet \exists (q, \sigma, t) \in \delta_c \Leftrightarrow \nexists (q, \sigma', t') \in \delta_u$$

  that is, the transitions leaving a given state are all either controllable or uncontrollable;
- $q_0 \in Q$ *is the initial state;*
- $A \subset Q$ *is the set of accepting states;*
- $B \subset Q$ *is the set of failure states, with* $A \cap B = \emptyset$ *and*

  $$\forall b \in B \bullet (b, \sigma, q) \in \delta_c \cup \delta_u \Rightarrow q = b.$$

The semantics of a BRG is its set of infinite executions, that is,

$$[\![brg]\!] = \{q_0, \sigma_0, q_1, \cdots \in (Q \times \Sigma)^\omega \mid \forall i \leq 0 \bullet (q_i, \sigma, q_{i+1}) \in \delta_c \cup \delta_u\}.$$

We now give an intuition of how a set of MSDs can be transformed into a BRG. The state space Q is defined as the cartesian products of the set of cuts of all the MSDs plus one self-looping failure state, which represents the

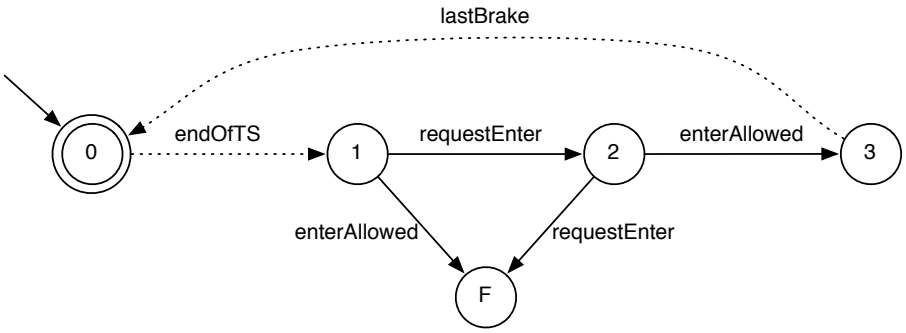


Figure 11.3: The BRG of the switch controller.

occurrence of a safety violation. There is only one accepting state, i.e. the state where all MSDs are inactive, which is also the initial state. The transitions between states are determined according to the enabled messages.[1] If at least one system message is enabled, the transitions leaving the current state are all controllable and correspond to the sending of an enabled system message. If sending a message results in a safety violation then the corresponding transition targets the failure state. Otherwise, it leads to the state where the MSD where the message is enabled have advanced to the next cut, while the other MSDs have not moved. If only environment messages are enabled, the outgoing transitions of the current state are uncontrollable. The state reached by such a transition is determined according to the same rules as in the controllable case.

Example 11.2.1 *Figure 11.3 depicts the BRG built from the MSD specification shown in Figure 11.1. The only initial and accepting state corresponds to the initial cut of the MSD. From there, the system can only move to state 1 after the environment message* `endOfTS` *occurs. Then, the system can send either* `requestEnter` *or* `enterAllowed`*. According to the MSD, only the former should be sent; the latter would thus result in a safety violation (modelled by state F). Once requestEnter occurs, the system reaches state 2 and can again send the same two messages. This time, only* `enterAllowed` *is permitted. Upon the sending of this message, state 3 is reached. Finally, the environment sends the message* `lastBrake`*, thereby making the system go back to the initial state and deactivating the MSD.*

To leverage these concepts to SPLs, we first have to extend BRG with variable transitions in the same way as FTS were defined.

[1] Actually, the alphabet is the set of message names combined with the name of the sending and receiving objects. We sometimes omit the object names in the transition label when the message name is sufficient.

Definition 11.4 (Featured Büchi Reachability Game)
A Featured Büchi Reachability Game (BRG) is a tuple $fbrg = (Q, \Sigma, \delta_{c,}, \delta_u, q_0, A, B, fm, \gamma)$ *where* Q, Σ, $\delta_{c,}$, δ_u, q_0, A, *and* B *are defined as in Definition 11.3,* fm *is an FM over a set of features* F, *and* $\gamma : (\delta_c \cup \delta_u) \to 2^{(2^F)}$ *is a total function that associates transitions with feature expressions.*

The projection of an FBRG is defined similarly to that of FTS.

Definition 11.5 (Projection of FBRG)
Let $fbrg = (Q, \Sigma, \delta_{c,}, \delta_u, q_0, A, B, fm, \gamma)$ *be an FBRG, an* $p \in [\![fm]\!]$ *be a product. The projection of* $fbrg$ *onto* p *is the BRG* $fbrg_{|p} = (Q, \Sigma, \delta'_{c,}, \delta'_u, q_0, A, B)$ *where*

$$\delta'_c = \{(q, \sigma, q') \in \delta_c \mid p \models \gamma(q, \sigma, q')\}$$
$$\delta'_u = \{(q, \sigma, q') \in \delta_u \mid p \models \gamma(q, \sigma, q')\}.$$

The semantics of an FBRG is then defined as a function that associates all valid products of fm to their respective projection.

Definition 11.6 (FBRG Semantics)
Let $fbrg$ *be an FBRG over an FM* fm. *The semantics of* $fbrg$ *is a total function* $[\![fbrg]\!] : [\![fm]\!] \to 2^{(Q \times \Sigma)^\omega}$ *such that*

$$\forall p \in [\![fm]\!] \bullet [\![fbrg]\!](p) = [\![fbrg_{|p}]\!].$$

Now that the formal model is defined, we show how to transform the scenario-based specification of an SPL as defined above into an FBRG. The states of the automaton result from the BRG transformation defined above applied to the union of the sets of MSD associated to the features. The idea is that all these MSDs are part of the specification regardless of the considered product, and that features restrict the *activation* (as opposed to the existence) of their associated MSDs. The occurrence of a message that triggers no MSD activation results in a single transition in the FBRG, which is executable by all products. When some MSDs should be activated (named *candidate* MSDs), multiple transitions are created, namely one per

subset of the set of features associated to these MSDs. Formally, let M be the set of newly activated MSDs. Then for each $M' \subseteq M$ a transition t is created, targets the state corresponding to the activation of the MSDs in M', and such that $\gamma(t)$ is defined as

$$\gamma(t) = \bigwedge_{m' \in M'} msdf(m') \wedge \bigwedge_{m \in M \setminus M'} \neg msdf(m)$$

where $msdf$ is the scenario-based specification of the SPL. Intuitively, $\gamma(t)$ is the conjunction of all the feature expressions associated to activated MSDs with the negation of all the feature expressions associated to a candidate MSD that has not been activated. Accordingly, a given product is able to execute only one of these transitions, i.e. the one corresponding to the set of activated MSDs that are part of this product's specification. Once the FBRG is built, it acts as the input to our synthesis algorithms with the aim to derive a controller for each product.

Example 11.2.2 *Consider an SPL composed of two products:* $\{r\}$ *and* $\{r, f\}$ *where* r *is the root feature and* f *is an optional feature. We associate the MSD* `RequestEnterAtEndOfTrackSection` *shown in Figure 11.1 to* r *and the MSD* `RC2CoordinateSwitchEntry` *(which is similar to RC1-CoordinateSwitchEntry in Figure 11.2 except that* `rc1` *and* `rc2` *have inverted roles) to feature* f*. Accordingly, the state space of the underlying FBRG consists of couples* (x, y) *where* x *(respectively* y*) is the current cut of* `RequestEnterAtEndOfTrackSection` *(respectively* `RC2CoordinateSwitchEntry`*).*

Now assume that the FBRG is currently in state $(2, 0)$*, that is, after* `endOfTS` *and* `requestEnter` *are sent. Figure 11.4 shows the outgoing transitions of this state. As in Example 11.2.1, a transition to the failure state occurs should the system send* `requestEnter` *a second time. The other transitions all involve the message* `enterAllowed`*. Their target state depends on the parameter given to the message as well as the enabled features. If the parameter is* $false$*, the FBRG reaches state* $(3, 0)$ *since the first message of* `RC2CoordinateSwitchEntry` *requires this parameter to be true. Otherwise, the MSD becomes active provided that the associated feature is enabled. This results in two transitions: one to state* $(3, 1)$ *labelled with* f *(modelling that the MSD activates), and the other to state* $(3, 0)$ *(which corresponds to the specification of the product without* f *as presented in Figure 11.3).*

11.3 Synthesis Algorithms

Now that we have defined FBRG and shown how to construct it from an MSD specification, we focus on the synthesis algorithms. We begin by presenting the single-system synthesis algorithms that we subsequently extend to support variability.

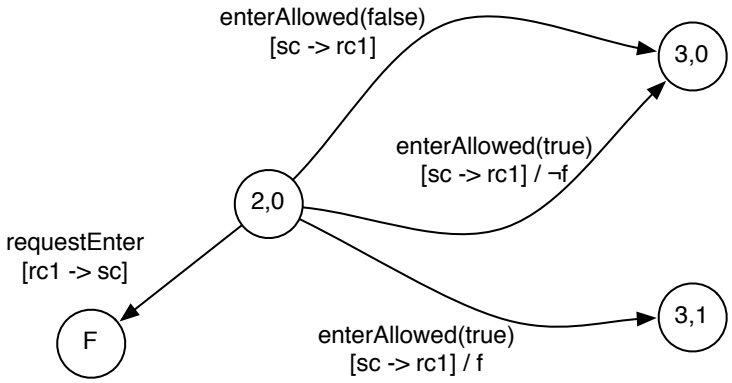


Figure 11.4: Computation of feature expressions within a FBRG.

11.3.1 Single-System Synthesis

Cassez *et al.* [35] proposed an algorithm to synthesize a discrete controller from a timed reachability game, itself being an extension of [136]. Given that we do not consider real time, our starting point is an untimed variant of this algorithm. As a reminder, the objective for the system is to fulfil the so-called *winning condition*, that is, to visit an accepting state infinitely often while avoiding the absorbing failure state. Accordingly, the objective of this algorithm is to compute the set of *winning* states, i.e. the states from which the system can guarantee to reach an accepting state infinitely often. We name *goal* states any state that is both accepting and winning. Thus, an arbitrary state is winning if and only if it can reach a goal state.

Computing the winning states comes down to a reachability analysis. The principles of Cassez *et al.*'s algorithm is to determine the set of goal states through a fixed-point computation starting with the complete set of accepting states, and iteratively removing the accepting states from which the system cannot guarantee to reach a goal state. Algorithm 6 gives an overview of this computation. Initially, the set G of goal states is the set of accepting states (Line 1). A first step, encapsulated in function *ReachGoal*, consists in computing the set Win all states from which the system can guarantee to reach a goal state (Line 2). Then at each iteration (Lines 3–5), we remove from G all goal states that cannot reach some goal state, i.e. that are not in Win. We iterate until no more state is removed from G; in this case, from any goal state in G the system can guarantee to reach a goal state (possibly the starting goal state itself), and thus to visit a goal state infinitely often. At that point, the set Win contains only all the states that can reach a goal state. If $q_0 \in Win$ then the specification is consistent as it means that the system can satisfy the winning condition from it. Given that set of winning states, a controller for the system is obtained by pruning

Input: $brg = (Q, \Sigma, \delta_c,, \delta_u, q_0, A, B)$.
Output: Win, the set of winning states.

```
1 G ← A;
2 Win ← ReachGoal(brg, G);
3 while G ≠ G ∩ Win do
4 |   G ← G ∩ Win;
5 |   Win ← ReachGoal(brg, G);
6 end
7 return Win
```

Algorithm 6: Synthesis(brg)

the non-winning states from the BRG.

Let us now have an in-depth look at the *ReachGoal* procedure. The difference with a standard reachability procedure is that it must distinguish between controllable and uncontrollable transitions. Remember that the outgoing transitions of a given state are either all controllable or all uncontrollable. Intuitively, in the controllable case the system can select which transition to execute. Therefore, a state q is winning if and only if it has at least one controllable transition leading to a winning state or a goal state, that is,

$$q \in Win \Leftrightarrow \bigvee_{(q,\sigma,q') \in \delta_c} (q' \in Win \cup G).$$

In the uncontrollable case, however, all the outgoing transitions must lead to a winning state or a goal state since the system has no control over their execution:

$$q \in Win \Leftrightarrow \bigwedge_{(q,\sigma,q') \in \delta_u} (q' \in Win \cup G).$$

Algorithm 7 formalises the *ReachGoal* procedure. During its execution, we maintain a set $Visited$ of visited states, a set $Waiting$ of transitions to explore, and a function *Depend* that associates a state with a subset of its incoming transitions (Lines 1–3). At each iteration, we remove a transition from $Waiting$ and analyse it (Line 5). The analysis splits into two parts depending on whether the target state is reached for the first time (Lines 6–13) or not (Lines 14–25). The first part is the *exploration*. The target state is added to the $Visited$ set and its outgoing transitions are added to $Waiting$ in order to pursue the exploration further. If the target state is a goal state then the source state is possibly a winning state. Hence, we re-put the transition into the $Waiting$ set in order to trigger the second part of the loop, i.e. the *re-evaluation*. Therein, we determine whether the source state of the transition is a winning state, depending on whether its outgoing transitions are controllable (Lines 15–17) or uncontrollable (18–20). If the state is found to be winning for the first time, we also have

Input: $brg = (Q, \Sigma, \delta_c, , \delta_u, q_0, A, B)$, $G \subseteq A$.
Output: Win, the set of states that can reach a state in G.

$Visited \leftarrow \{q_0\}$;
$Waiting \leftarrow \{(q_0, \sigma, q) \in \delta_c \cup \delta_u\}$;
$Depend(q_0) \leftarrow \emptyset$;
while $Waiting \neq \emptyset$ **do**
  Take $t = (q, \sigma, q') \in Waiting$;
  if $q' \notin Visited$ **then**
    $Visited \leftarrow Visited \cup \{q'\}$;
    $Depend(q') \leftarrow \{t\}$;
    $Waiting \leftarrow Waiting \cup \{(q', \sigma, q'') \in \delta_c \cup \delta_u\}$;
    if $q' \in G$ **then**
      $Waiting \leftarrow Waiting \cup \{t\}$;
    end
  end
  else
    if $t \in \delta_c$ **then**
      $Win^* \leftarrow \bigvee_{(q,\sigma'',q'') \in \delta_c} (Win(q'') \cup G)$;
    end
    else
      $Win^* \leftarrow \bigwedge_{(q,\sigma'',q'') \in \delta_u} (Win(q'') \cup G)$;
    end
    if $Win^* \wedge \neg Win(q)$ **then**
      $Win(q) \leftarrow Win^*$;
      $Waiting \leftarrow Waiting \cup Depend(q)$;
    end
  end
end
return Win

Algorithm 7: ReachGoal(brg, G)

to re-evaluate its previously visited predecessors; we thus add its incoming transitions contained in *Depend* to the $Waiting$ set in order to trigger these re-evaluations (Lines 21-24). Once all the transitions have been explored and re-evaluated as needed, the algorithm returns the set of winning states (Line 28).

11.3.2 Mass Synthesis

Our next objective is to turn the above synthesis algorithms into variability-aware extensions. To that aim, we revisit the concepts introduced in the previous section and identify which ones have to be redefined to consider a set of products as opposed to a single product. The most important definition, which determines whether the specification is consistent, is the winning condition: a state is winning if and only if from this state the system can guarantee to reach an accepting state infinitely often. In FBRG, variability impacts the executability of transitions, and thus the reachability between states. Therefore, a state can be winning for only a subset of the valid products, and the set of winning states Win is replaced by a function from Q to 2^{2^F} that associates a given state to the feature expression encoding the products for which the state is winning. The definition of goal states is modified similarly, since a goal state is an accepting state that is also winning. For recall, Algorithm 6 determines the set of goal states through a greatest fixed-point computation, starting with the set of accepting states and potentially removing states from the G set at each iteration. In our extension (see Algorithm 8), we define G as a function from A to 2^{2^F} that associates to each accepting state the feature expression encoding the set of products for which this state is also winning. We compute it again via fixed-point computation, starting with $G(a) = fm$ for any $a \in A$ (that is, accepting states are goal states for all valid products). At each iteration, we compute the function Win according to the current function G, and updates G accordingly. The fixed-point is not reached if and only if after an iteration an accepting state is a goal state for a smaller (possibly empty) set of products.

It remains to detail the procedure to compute the winning function Win, which is formalised in Algorithm 9. Without surprise, it is very similar to its single-system counterpart. In the initialisation, the only difference is that every state is considered winning for no product (Lines 1–3). The first half of the loop (Lines 9–16) is completely equivalent to that of Algorithm 9. The second half is where variability has an impact. When determining the winning condition for a given state, we consider again the controllable case and the uncontrollable case separately. Figure 11.5 illustrates the winning conditions in both cases. In the controllable case (Lines 18–20), state q is winning for a product p if and only if there exists a transition from q available to p leading to a state which is a winning state or a goal state for

Input: $fbrg = (Q, \Sigma, \delta_c,, \delta_u, q_0, A, B, fm, \gamma)$.
Output: Win, the set of winning states.

```
1  foreach a ∈ A do
2  |  G(a) ← fm;
3  end
4  Win ← FReachGoal(fbrg, G);
5  while ∃a ∈ A • G(a) ⇏ Win(a) do
6  |  foreach a ∈ A do
7  |  |  G(a) ← Win(a);
8  |  end
9  |  Win ← FReachGoal(fbrg, G);
10 end
11 return Win
```

Algorithm 8: F-Synthesis($fbrg$)

p, that is,

$$p \models Win(q) \Leftrightarrow \exists (q, \sigma, q') \in \delta_c \bullet p \in [\![\gamma(q, \sigma, q')]\!] \cap ([\![Win(q')]\!] \cup [\![G(q')]\!]).$$

If we apply the above formula to the whole set of products, we obtain that the set of products for which q is winning via q' is encoded by the conjunction of γ with the disjunction of $Win(q')$ and $G(q')$. We thus have

$$Win(q) \Leftrightarrow \bigvee_{(q,\sigma'',q'') \in \delta_c} \big(\gamma(q, \sigma'', q'') \wedge (Win(q'') \vee G(q''))\big).$$

The uncontrollable case is more subtle. A state q is winning for product q if and only if all the following conditions hold:

1. At least one transition from q is available to p.
2. All the transitions from q *available to* p lead to a state which is a winning state or a goal state for p.

Accordingly, the products for which p is winning are captured by the feature expression

$$\Big(\bigvee_{(q,\sigma'',q'') \in \delta_u} \gamma(q, \sigma'', q'')\Big) \wedge \bigwedge_{(q,\sigma'',q'') \in \delta_u} \big(\gamma(q, \sigma'', q'') \Rightarrow Win(q'') \vee G(q'')\big).$$

In the second half of the loop of Algorithm 9, we apply the above two formulae to compute the set of products for which a state is winning (Lines 18–23). If this set has changed with respect to prior computations, the algorithm re-evaluates the winning condition for the predecessors of q that it visited earlier (Lines 24–27).

Input: $brg = (Q, \Sigma, \delta_c,, \delta_u, q_0, A, B)$, $G \subseteq A$.
Output: Win, the set of states that can reach a state in G.

foreach $q \in Q$ **do**
  $Win(q) = \bot$;
end
$Visited \leftarrow \{q_0\}$;
$Waiting \leftarrow \{(q_0, \sigma, q) \in \delta_c \cup \delta_u\}$;
$Depend(q_0) \leftarrow \emptyset$;
while $Waiting \neq \emptyset$ **do**
  Take $t = (q, \sigma, q') \in Waiting$;
  if $q' \notin Visited$ **then**
    $Visited \leftarrow Visited \cup \{q'\}$;
    $Depend(q') \leftarrow \{t\}$;
    $Waiting \leftarrow Waiting \cup \{(q', \sigma, q'') \in \delta_c \cup \delta_u\}$;
    if $q' \in dom(G)$ **then**
      $Waiting \leftarrow Waiting \cup \{t\}$;
    end
  end
  else
    if $t \in \delta_c$ **then**
      $Win^* \leftarrow \bigvee_{(q,\sigma'',q'') \in \delta_c} \big(\gamma(q, \sigma'', q'') \wedge (Win(q'') \vee G(q''))\big)$;
    end
    else
      $Win^* \leftarrow \big(\bigvee_{(q,\sigma'',q'') \in \delta_u} \gamma(q, \sigma'', q'')\big) \wedge$
      $\bigwedge_{(q,\sigma'',q'') \in \delta_u} \big(\gamma(q, \sigma'', q'') \Rightarrow Win(q'') \vee G(q'')\big)$;
    end
    if $Win^* \not\Leftrightarrow Win(q)$ **then**
      $Win(q) \leftarrow Win^*$;
      $Waiting \leftarrow Waiting \cup Depend(q)$;
    end
  end
end
return Win

Algorithm 9: ReachGoal(brg, G)

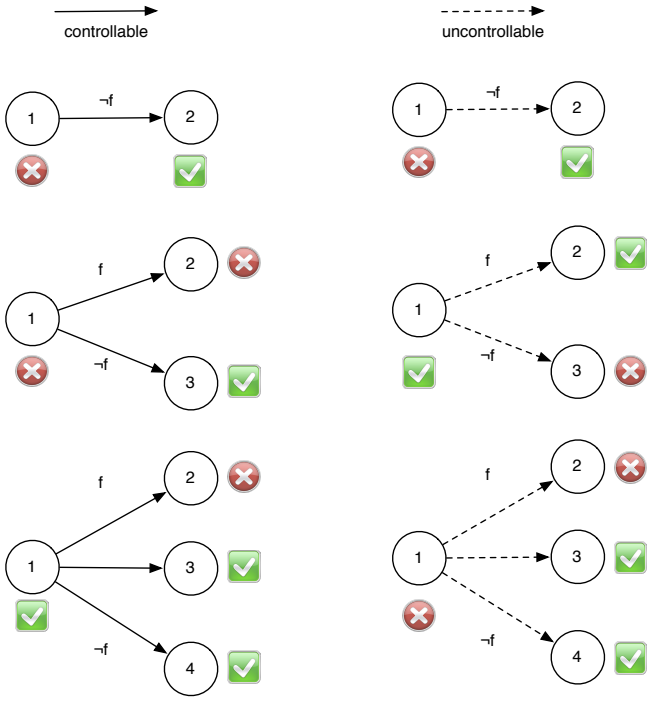


Figure 11.5: Winning conditions for a product $p = \{f\}$.

11.4 Perspectives

After applying the algorithm to compute the winning function, two cases may happen. If the initial state q_0 is not winning for at least one valid product (i.e., $[\![Win(q_0)]\!] \cap [\![fm]\!] = \emptyset$), it means that the SPL specification is inconsistent for all products and thus no controller can be synthesized. Otherwise, we can extract from the FBRG a controller for any product p satisfying $Win(q_0)$. This controller is obtained by performing the following steps:

1. Remove every state q in the FBRG such that p does not satisfy $Win(q)$ as well as its incoming and outgoing transitions;

2. Compute the projection of the resulting FBRG.

Instead of a controller for an individual product, we can keep a concise representation for the controllers of all the products. To that aim, we have to make each state inaccessible to the products for which the state is not winning. This is achieved by replacing the feature expression labelling any incoming transition of each state q by its conjunction with $Win(q)$. In this case, the products have access only to the states that are winning for them. Also, products for which the specification is inconsistent should not even

be produced; one can thus add a constraint in the FM that makes these products invalid. This variability-aware controller can be regarded as an FTS. It is therefore possible to further analyse the controller using the FTS model-checking algorithms, thereby determining the relevant properties of each product for which the specification is consistent. This combination of synthesis and model checking offers interesting application perspectives, as they provide together a complete approach ranging from an intuitive modelling language to the verification of critical behavioural requirements.

Part VI

Postface

Chapter 12

Conclusion

The initial objective of this thesis was to improve the efficiency and expressiveness of SPL model checking. Beforehand, we had to analyse the different available modelling formalisms, and to determine which of them was the more appropriate to act as the basis of our work. FTS appeared as the ideal candidate: they were made with the specific intent of designing SPL verification techniques, and they revealed to hold the appropriate level of expressiveness together with efficient, dedicated verification algorithms. Moreover, they are combined with a feature model, the most common language to represent variability. Our goal thus became to extend the FTS theory with new formalisms and algorithms. FTS being a generalization of TS, it appeared natural to study how single-system model checking improved to reach its current level of optimisation, and extended to provide modelling formalism for a broad range of systems. This was the first source of inspiration for our work. However, designing similar improvements for FTS is far from being a straightforward application of the techniques developed for single systems. Indeed, variability is arbitrarily invasive and cannot be considered orthogonally. This is the reason why we had to study such combinations in depth.

We first addressed the problem of optimising FTS model checking (Objective **O1**). Concretely, we aimed to design abstraction methods that can reduce verification time. A prerequisite was the definition of simulation relation for SPLs. This relation allows the comparison between models with respect to the satisfaction of properties. In our case, the fact that an FTS simulates another means that the products satisfying an LTL formula in the former also satisfy it in the latter. Our FTS simulation relation is the perfect example that variability has a deep impact on behaviour; its definition intensively involves the feature expressions labelling the transitions (see Definition 5.3). Having FTS simulation as a sound basis for behavioural comparison, we next studied how to design FTS abstractions. We first attempted to perform abstraction on the state space of the FTS and its vari-

ability separately. This led to two families of abstractions: one reducing the number of states (state abstraction), and one raising the number of products able to execute variable transitions (feature abstraction). Although variability does not seem to impact the former, it still drives the refinement process: before each refinement, the products for which the model checker has correct results are not verified anew during the subsequent verifications. Our experiments showed that the two families of abstraction have completely different impacts on verification time. Procedures based on feature abstraction generally result in verifying the original model; still, in a minority of cases – in particular when the property is satisfied by all products – its application yields drastic reductions in verification time, and thus overcomes the exponential blow-up due to variability. On the contrary, state abstraction improves performance in most cases, but not as significantly as feature abstraction; in the other cases, however, it can yield poor performance results. We also attempted to design an abstraction that mixes the above two. The results of its evaluation follow the same trend as those of state abstraction, as if this abstraction cancelled the effect of feature abstraction. We believe, however, that **other mixed abstractions** could be designed based on the topology of the FTS and its variability. This is an interesting direction for future work.

Abstraction is certainly not the only way to reduce verification time. When the system is composed of multiple components that evolve in parallel, the model ultimately verified is the parallel composition of models of these components. A common technique to reduce the number of executions to check in the parallel composition is the **partial-order reduction**. In a nutshell, it consists in fixing the order in which the components execute their next steps when this order has no impact on the overall behaviour of the system. Studying this technique for FTS is interesting and not straightforward, as the condition on the order can depend on the features. **Compositional reasoning** is another interesting perspective to reduce verification time. The idea is to analyse the behavioural impact of each feature separately so as to infer whether a feature leads to the violation of a given property. It was already studied for CTL model checking [91, 133, 132, 134, 135]. We preliminary worked on an equivalent approach for LTL [67], but it is limited to conservative features. The idea is to check whether the portion of state space added by the feature created a loop including an accepting state. Designing such a method for other features is, however, more challenging, and its efficiency and applicability still need to be assessed.

After efficiency, we focused on expressiveness (Objective **O2**). We were first concerned with augmenting the type of behaviour that the SPL products can exhibit. In particular, we addressed the modelling of real-time behaviour and designed a new formalism (*viz.* FTA). Then we added the support for non-Boolean features. These two increases in expressiveness revealed that the major challenge is to find the appropriate data structures to

encode states and sets of products. In our FTA implementation, we encoded real-time and variability separately, using DBMs and BDDs, respectively. An alternative, which we did not further explored, is to design a **combined representation** that can efficiently check inclusion in terms of time zone and set of products. Should we tackle the modelling of other kinds of SPL behaviour (e.g., **stochastic behaviour**), we would face similar challenges. Non-boolean features present two other substantial difficulties. First, the support for attributes through SMT solvers had a significant impact on verification time. We proposed an alternative based on Boolean conversion, but its implementation highly depends on the modelling language. There is thus a need for finding a generic solution to diminish the overhead of handling attributes. Second, although they could be combined with the state-of-the-art FTS formalism and algorithms, there is currently no guarantee that they are compatible with all the extensions reported in this thesis. For instance, the decidability of checking an FTA where transition delays varies in function of the value of feature attributes is not guaranteed. Additional hypotheses on the impact of variability would likely be required.

An important future work, which we believe should be one of the main focus in SPL verification for the coming years, is to make all these theoretical developments available to engineers. To make a step towards that direction, we implemented ProVeLines, a new model checker for SPLs (Objective **O3**). Given that all our work is based on extensions of FTS, it seemed natural to design ProVelines following the principles of SPL engineering. However, the continuous addition of features has made its maintainability increasingly complex. Furthermore, just like in SNIP, the algorithms implemented within ProVeLines are tightly coupled to its input language, viz. fPromela. This makes the addition of new, industry-specific modelling languages difficult. To overcome this limitation, we recently started a re-engineering of ProVeLines' architecture [70]. Our ultimate objective is to provide a **customizable tool** that can be plugged into a real-world industrial process. Our current perspectives include the integration of our tool with variability-aware state machines and a code generator. This would allow one to model the behaviour of the system in a language accepted by industrials, to verify the underlying FTS against desired properties, and then to automatically generate the source code controlling the modelled behaviour of the system. By construction, this source code satisfies the verified properties and future bugs will be due to the customized (i.e. not generated) part of the code.

As a last objective (**O4**), we successfully applied the principles of FTS model checking to other formal verification methods, *viz.* adaptive systems and controller synthesis. In addition to their individual contribution, the corresponding chapters serve a more general purpose. In Chapter 10, we showed that the combination of variability and behaviour modelling can be used in other domains than SPL engineering. Furthermore, from our study on controller synthesis (see Chapter 11) we can derive basic principles to

generalise formal verification methods to variability-intensive systems:

1. **Identify the mathematical concepts influenced by variability.** For FTS, these concepts are the satisfaction of formulae and the executability of transitions. In FBRG, the executability of transitions is also subject to variability, as is the winning condition.

2. **Update the computation formulae.** The satisfaction of LTL formulae depends on the state reachability relation, which in turn is determined by the transitions. As for the winning condition in FBRG, it can be computed on each individual state in function of its outgoing transitions, their controllability, and their variability.

3. **Extend the algorithms.** The last step is to rewrite the single-system algorithms according to the new concepts and computation formulae introduced previously. This may require to extend additional concepts with variability, e.g. reachable states in FTS and goal states in FBRG. The termination condition also changes: in FTS, we have to visit all states with all the products able to reach them, whereas the fixed-point in the synthesis algorithm is reached when there is no goal state whose associated set of products has been reduced.

Every development we made can be regarded in a particular application of the above procedure. The most important contribution of this thesis could be to have shown that the principles of FTS model checking can be transposed to other areas. Our dearest hope is that our work will constitute a source of inspiration for variability-aware formal methods that will be developed in the future, thereby contributing to our predecessor's objective to make FTS the state-of-the-art formalism for SPL verification [51]. That is, we hope to have made the first steps to enable *formal methods for the masses*.

Bibliography

[1] BTC EmbeddedValidator. http://www.btc-es.de/index.php?lang=2&idcatside=5.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Composing feature models. In *SLE'09*, pages 62–81, 2009.

[3] Rasmus Adler, Ina Schaefer, Tobias Schuele, and Eric Vecchié. From model-based design to formal verification of adaptive embedded systems. In *Proceedings of the formal engineering methods 9th international conference on Formal methods and software engineering*, Proceedings of ICFEM '07, pages 76–95, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, pages 672–678, 2012.

[5] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of FASE '98*, pages 21–37, Lisbon, Portugal, March 1998.

[6] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastian Uchitel. Learning operational requirements from goal models. In *ICSE 31*, pages 265–275, 2009.

[7] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Inf. Comput.*, 104:2–34, May 1993.

[8] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–203, January 1994.

[9] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, September 2002.

[10] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *Proceedings of the 9th In-*

ternational Conference on Concurrency Theory, CONCUR'98, pages 163–178, London, UK, 1998. Springer-Verlag.

[11] M. Antkiewicz and K. Czarnecki. Featureplugin: Feature modeling plug-in for eclipse. In *OOPSLA'04*, 2004.

[12] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Feature-interaction detection using feature-aware verification. In *ASE'11*, pages 372–375. IEEE, 2011.

[13] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *ICSE'13*, pages 482–491, 2013.

[14] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical computer science*, 303(1):7–34, 2003.

[15] Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi. Formal description of variability in product families. In *SPLC'11*, pages 130–139. Springer-Verlag, 2011.

[16] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[17] Don S. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC'05*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[18] Basil Becker and Holger Giese. Modeling of correct self-adaptive systems: a graph transformation system based approach. In *Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology*, Proceedings of CSTST '08, pages 508–516, New York, NY, USA, 2008. ACM.

[19] Richard Bellman. A markovian decision process. *Indiana University Mathematics Journal*, 6:679–684, 1957.

[20] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, September 2010.

[21] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. Fama: Tooling a framework for the automated analysis of feature models. In *VaMoS'07*, pages 129–134, 2007.

[22] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and*

Control of Hybrid Systems III, number 1066 in LNCS, pages 232–243. Springer–Verlag, October 1995.

[23] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *TACAS'96*, pages 431–434. Springer-Verlag, 1996.

[24] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.

[25] Danilo Beuche. Modeling and building software product lines with pure: :variants. In *SPLC'08*, page 358, 2008.

[26] Dirk Beyer. Second competition on software verification - (summary of sv-comp 2013). In *TACAS '13*, pages 594–609, 2013.

[27] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV '11*, pages 184–190, 2011.

[28] Y. Bontemps and P. Heymans. From live sequence charts to state machines and back: A guided tour. *Transactions on Software Engineering*, 31(12):999–1014, 2005.

[29] Quentin Boucher, Andreas Classen, Patrick Heymans, Arnaud Bourdoux, and Laurent Demonceau. Tag and prune: A pragmatic approach to software product line implementation. In *ASE'10*, pages 333–336. ACM, 2010.

[30] Glenn Bruns. A practical technique for process abstraction. In *Proceedings of the 4th International Conference on Concurrency Theory*, CONCUR '93, pages 37–49, London, UK, 1993. Springer-Verlag.

[31] Glenn Bruns and Patrice Godefroid. Model checking with multi-valued logics. In *ICALP '04*, pages 281–293, 2004.

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

[33] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comp.*, 98(2):142–170, 1992.

[34] M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[35] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR '05*, pages 66–80. Springer, 2005.

[36] Franck Cassez, Mark Dermot Ryan, and Pierre yves Schobbens. Proving feature non-interaction with alternating-time temporal logic. In *Language Constructs for Describing Features – proceedings of the FIREworks workshop*. Springer, 2001.

[37] K. Chatterjee, L. Doyen, and T.A. Henzinger. A survey of partial-observation stochastic parity games. *Formal Methods in System Design*, pages 1–17, 2012.

[38] Marsha Chechik, Benet Devereux, Steve M. Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Trans. Softw. Eng. Methodol.*, 12(4):371–408, 2003.

[39] Marsha Chechik, Benet Devereux, and Arie Gurfinkel. Model-checking infinite state-space systems with fine-grained abstractions using spin. In *SPIN '01*, pages 16–36, 2001.

[40] Marsha Chechik, Steve M. Easterbrook, and Victor Petrovykh. Model-checking over multi-valued logics. In *FME '01*, pages 72–98, 2001.

[41] W. Chen and M. Saif. Observer-based fault diagnosis of satellite systems subject to time-varying thruster faults. *Journal of dynamic systems, measurement, and control*, 129(3):352–356, 2007.

[42] Betty H. C. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for Self-Adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, chapter 1, pages 1–26. Springer Berlin / Heidelberg, 2009.

[43] R. Cieslak, C. Desclaux, A.S. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *Automatic Control, IEEE Transactions on*, 33(3):249–260, 1988.

[44] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV '02*, volume 2404 of *LNCS*. Springer, July 2002.

[45] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[46] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[47] 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.*, 8:244–263, April 1986.

[48] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Emerson and Aravinda Sistla, editors, *Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer Berlin / Heidelberg, 2000.

[49] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ansi-c programs using sat. *Form. Methods Syst. Des.*, 25(2-3):105–127, September 2004.

[50] Andreas Classen. Modelling with fts: a collection of illustrative examples. Technical report, University of Namur (FUNDP), 2010.

[51] Andreas Classen. *Modelling and Model Checking Variability-Intensive Systems (Ph. D. dissertation)*. PhD thesis, University of Namur (FUNDP), 2011.

[52] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *SCP*, 76:1130–1143, December 2011.

[53] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.

[54] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming*, 80:416–439, 2014.

[55] Andreas Classen, Maxime Cordy, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Snip: An efficient model checker for software product lines. Technical report, University of Namur (FUNDP), 2011.

[56] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their

application to LTL model checking. *Transactions on Software Engineering*, pages 1069–1089, 2013.

[57] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE'08*, volume 4961 of *LNCS*, pages 16–30. Springer, 2008.

[58] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *ICSE'11*, pages 321–330. ACM, 2011.

[59] 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 *ICSE'10*, pages 335–344. ACM, 2010.

[60] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[61] Consultative Committee for Space Data Systems (CCSDS). *CCSDS File Delivery Protocol (CFDP): Blue Book, Issue 4*. NASA, 2007.

[62] Maxime Cordy, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking adaptive software with featured transition systems. In Javier Camara, Rogerio Lemos, Carlo Ghezzi, and Antonia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *LNCS*, pages 1–29. Springer Berlin Heidelberg, 2013.

[63] Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Managing evolution in software product lines : A model-checking perspective. In *VaMoS'12*, pages 183–191. ACM, 2012.

[64] Maxime Cordy, Andreas Classen, Gilles Perrouin, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Simulation-based abstractions for software product-line model checking. In *ICSE'12*, pages 672–682. IEEE, 2012.

[65] Maxime Cordy, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Behavioural modelling and verification of real-time software product lines. In *SPLC'12*. ACM, 2012.

[66] Maxime Cordy, Axel Legay, Pierre-Yves Schobbens, and Louis-Marie Traonouez. A framework for the rigorous design highly adaptive timed systems. In *FormaliSE'13*, pages 64–70. IEEE, 2013.

[67] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Towards an incremental automata-based approach for software product-line model checking. In *Proceedings of the 16th International Software Product Line Conference-Volume 2*, pages 74–81. ACM, 2012.

[68] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *ICSE'13*, pages 472–481. IEEE, 2013.

[69] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Provelines: A product-line of verifiers for software product lines. In *SPLC'13*, pages 141–146. ACM, 2013.

[70] Maxime Cordy, Marco Willemart, Bruno Dawagne, Patrick Heymans, and Pierre-Yves Schobbens. An extensible platform for product-line behavioural analysis. In *SPLaT'14 (to appear)*. ACM, 2014.

[71] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252, New York, NY, USA, 1977. ACM.

[72] William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[73] J.E.R. Cury, B.H. Krogh, and T. Niinomi. Synthesis of supervisory controllers for hybrid systems based on approximating automata. *Automatic Control, IEEE Transactions on*, 43(4):564–568, 1998.

[74] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, pages 422–437, 2005.

[75] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[76] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[77] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories at OOPSLA'05*, San Diego, California, USA, 2005. ACM, ACM.

[78] A. David, G. Behrmann, P. Bulychev, J. Byg, T. Chatain, K. G. Larsen, P. Pettersson, J. I. Rasmussen, J. Srba, W. Yi, K. Y. Joergensen, D. Lime, M. Magnin, O. H. Roux, and L.-. Traonouez. Tools for model-checking timed systems. In *Communicating Embedded Systems – Software and Design*, pages 165–225. ISTE Publ./John Wiley, 2009.

[79] S.M. Davis. *Future Perfect*. Addison-Wesley, 1987.

[80] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, pages 337–340. Springer-Verlag, 2008.

[81] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3):313–341, 2008.

[82] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[83] V. D'silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, July 2008.

[84] Charles E. Ebeling. *An introduction to reliability and maintainability engineering*. McGraw-Hill, 1997.

[85] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT'03*, pages 502–518. Springer, 2003.

[86] Rudiger Ehlers, Daniel Fass, Michael Gerke, and Hans-Jorg Peter. Fully symbolic timed model checking using constraint matrix diagrams. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 360–371, Washington, DC, USA, 2010. IEEE Computer Society.

[87] Stephan Falke, Florian Merz, and Carsten Sinz. The bounded model checker llbmc. In *ASE '13*, pages 706–709, 2013.

[88] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *ICSE '11*, pages 341–350, 2011.

[89] Emmanuel Filiot, Naiyong Jin, and Jean-Francois Raskin. Antichains and compositional algorithms for ltl synthesis. *Formal Methods in System Design*, 39:261–296, 2011. 10.1007/s10703-011-0115-3.

[90] Dario Fischbein, Sebastian Uchitel, and Victor Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA'06, ISSTA 2006 workshop*, pages 39–48. ACM Press, 2006.

[91] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 152–163, New York, NY, USA, 2001. ACM.

[92] Nissim Francez and Ira R. Forman. Superimposition for interacting processes. In *CONCUR'90*, volume 458 of *LNCS*, pages 230–245. Springer, 1990.

[93] Silvio Ghilardi and Silvio Ranise. Mcmt: A model checker modulo theories. In *IJCAR'10*, pages 22–29, 2010.

[94] Holger Giese. Modeling and verification of cooperative self-adaptive mechatronic systems. In *Proceedings of the 12th Monterey conference on Reliable systems on unreliable networked platforms*, pages 258–280, Berlin, Heidelberg, 2007. Springer-Verlag.

[95] Holger Giese and Betty H. C. Cheng, editors. *SEAMS '11: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, New York, NY, USA, 2011. ACM.

[96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

[97] Hassan Gomaa and Mohamed Hussein. Dynamic software reconfiguration in software product families. In Frank van der Linden, editor, *Proceedings of PFE '03*, volume 3014 of *LNCS*, pages 435–444. Springer Berlin / Heidelberg, 2004.

[98] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK, 1997. Springer-Verlag.

[99] Joel Greenyer, Amir Molzam Sharifloo, Maxime Cordy, and Patrick Heymans. Efficient consistency checking of scenario-based product line specifications. In *RE '12*, pages 161–170, 2012.

[100] Joel Greenyer, Amir Molzam Sharifloo, Maxime Cordy, and Patrick Heymans. Features meet scenarios: Modeling and consistency-

checking scenario-based product-line specifications. *Requirements Engineering Journal*, 18(2):175–198, 2013.

[101] N. Griffeth, Y.-J. Lin, and al. *Feature interactions in telecommunications and software systems (FIW, ICFI)*. Ios Press, 1992-2012.

[102] T. Gross and H. Sayama. *Adaptive Networks: Theory, Models and Applications*. Understanding Complex Systems. Springer, 2009.

[103] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and model checking software product lines. In *FMOODS'08*, pages 113–131. Springer, 2008.

[104] Dimitar P. Guelev, Mark Ryan, and Pierre Yves Schobbens. Model-checking the preservation of temporal properties upon feature integration. *Electron. Notes Theor. Comput. Sci.*, 128:311–324, May 2005.

[105] Dimitar P. Guelev, Mark Ryan, and Pierre Yves Schobbens. Synthesising features by games. *Electron. Notes Theor. Comput. Sci.*, 145:79–93, January 2006.

[106] Arie Gurfinkel and Marsha Chechik. Multi-valued model checking via classical model checking. In *CONCUR*, pages 263–277, 2003.

[107] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-class variability modeling in matlab/simulink. In *VaMoS '13*, pages 4:1–4:8, New York, NY, USA, 2013. ACM.

[108] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994.

[109] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[110] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Formal Methods in Software and Systems Modeling*, volume 3393/2005, pages 309–324. Springer-Verlag, 2005.

[111] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM)*, 7(2):237–252, May 2008.

[112] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, August 2003.

[113] Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. A code tagging approach to software product line development - an application to satellite communication libraries. *STTT*, 14(5):553–566, 2012.

[114] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[115] Arnaud Hubaux, Quentin Boucher, Herman Hartmann, Raphaël Michel, and Patrick Heymans. Evaluating a textual feature modelling language: four industrial case studies. In *SLE'10*, pages 337–356, Berlin, Heidelberg, 2011. Springer-Verlag.

[116] Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. In David Sands, editor, *Programming Languages and Systems*, volume 2028 of *Lecture Notes in Computer Science*, pages 155–169. Springer Berlin Heidelberg, 2001.

[117] Arno Jeanjot. High-level modelling and formal semantics of product-line behaviour. Master's thesis, University of Namur, Belgium, 2013.

[118] G. Kalyon, T. Le Gall, H. Marchand, and T. Massart. Symbolic supervisory control of infinite transition systems under partial observation using abstract interpretation. *Discrete Event Dynamic Systems*, pages 1–41, 2012.

[119] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.

[120] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *ICSE'08*, pages 311–320, New York, NY, USA, 2008. ACM.

[121] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244, 2005.

[122] Narges Khakpour, Saeed Jalili, Carolyn Talcott, Marjan Sirjani, and MohammadReza Mousavi. Formal modeling of evolving self-adaptive systems. *Science of Computer Programming*, September 2011.

[123] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, October 1990.

[124] Dexter Kozen. Results on the propositional μ-calculus. pages 348–359. 1982.

[125] J. Kramer and J. Magee. Analysing dynamic change in software architectures: A case study. In *CDS'98*, pages 91–100. IEEE Computer Society, 1998.

[126] J. Kramer, J. Magee, M. Sloman, and A. Lister. Conic: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1–10, 1983.

[127] Shriram Krishnamurthi and Kathi Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16, April 2007.

[128] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137–146, New York, NY, USA, 2004. ACM.

[129] Sandeep Kulkarni and Karun Biyani. Correctness of component-based adaptation. In Ivica Crnkovic, Judith Stafford, Heinz Schmidt, and Kurt Wallnau, editors, *Proceedings of CBSE '04*, volume 3054 of *LNCS*, pages 48–58. Springer Berlin / Heidelberg, 2004.

[130] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In *ESOP*, pages 64–79, 2007.

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

[132] Harry C. Li, Shriram Krishnamurthi, and Kathi Fisler. Interfaces for modular feature verification. In *Proceedings of ASE'02*, pages 195–204, 2002.

[133] Harry C. Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. In *SIGSOFT FSE*, pages 89–98, 2002.

[134] Harry C. Li, Shriram Krishnamurthi, and Kathi Fisler. Modular verification of open features using three-valued model checking. *Automated Software Engg.*, 12:349–382, July 2005.

[135] Jing Liu, Samik Basu, and Robyn R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engg.*, 18:39–76, March 2011.

[136] Xinxin Liu and ScottA. Smolka. Simple linear-time algorithms for minimal fixed points. In KimG. Larsen, Sven Skyum, and Glynn Winskel,

editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 53–66. Springer Berlin Heidelberg, 1998.

[137] Rupak Majumdar, Elaine Render, and Paulo Tabuada. Robust discrete synthesis against unspecified disturbances. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*, HSCC '11, pages 211–220, New York, NY, USA, 2011. ACM.

[138] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS 95*, pages 229–242. Springer, 1995.

[139] A. Markov. Extension of the Limit Theorems of Probability Theory to a Sum of Variables Connected in a Chain. In R. Howard, editor, *Dynamic Probabilistic Systems (Volume I: Markov Models)*, chapter Appendix B, pages 552–577. John Wiley & Sons, Inc., New York City, 1971.

[140] Raúl Mazo, Camille Salinesi, Daniel Diaz, and Alberto Lora-Michiels. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In *ENASE'11*, pages 188–199, 2011.

[141] Kenneth McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[142] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A formal semantics for feature cardinalities in feature diagrams. In *VaMoS'11*, pages 82–89, New York, NY, USA, 2011. ACM.

[143] Jean-Vivien Millo, S Ramesh, Shankara Narayanan Krishna, Ganesh Khandu Narwane, et al. Compositional verification of software product lines. In *10th International Conference on integrated Formal Methods*, 2013.

[144] Robin Milner. An algebraic definition of simulation between programs. Technical report, Stanford University, Stanford, CA, USA, 1971.

[145] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[146] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Handling unbounded loops with esbmc 1.20 - (competition contribution). In *TACAS*, pages 619–622, 2013.

[147] Malte Plath and Mark Ryan. Feature integration using a feature construct. *SCP*, 41(1):53–84, 2001.

[148] A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57, 1977.

[149] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software product line engineering - foundations, principles, and techniques.* Springer, 2005.

[150] Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *ASE'08*, pages 347–350. IEEE CS, 2008.

[151] D.J. Ragsdale, C.A. Carver Jr, J.W. Humphries, and U.W. Pooch. Adaptation techniques for intrusion detection and intrusion response systems. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 4, pages 2344–2349. IEEE, 2000.

[152] Reactive Systems. Reactis Tester. http://www.reactive-systems.com.

[153] Mathieu Sassolas, Marsha Chechik, and Sebastian Uchitel. Exploring inconsistencies between modal transition systems. *Softw. Syst. Model.*, 10:117–142, February 2011.

[154] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE'06*, pages 139–148, 2006.

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

[156] Maurice H. ter Beek, Franco Mazzanti, and Aldi Sulova. Vmc: A tool for product variability analysis. In *FM '12*, pages 450–454, 2012.

[157] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *CIT*, 10(1):1–18, 2002.

[158] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *LICS'86*, pages 332–344. IEEE CS, 1986.

[159] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. The pla model: on the combination of product-line analyses. In *VaMoS*, page 14, 2013.

[160] Farn Wang. Efficient model-checking of timed automata with clock-restriction diagram. In *APLAS*, pages 207–224, 2001.

[161] Marco Willemart. Automated verification of software product lines: An industrial case study. Master's thesis, University of Namur, Belgium, 2013.

[162] H. Wong-Toi. The synthesis of controllers for linear hybrid automata. In *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, volume 5, pages 4607–4612. IEEE, 1997.

[163] H. Wong-Toi and G. Hoffmann. The control of dense real-time discrete event systems. In *Decision and Control, 1991., Proceedings of the 30th IEEE Conference on*, pages 1527–1528. IEEE, 1991.

[164] Wang Yi. Real-time behaviour of asynchronous agents. In *Proceedings of CONCUR '90*, pages 502–520, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[165] Pamela Zave and Michael A. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.

[166] Ji Zhang and Betty H.C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, 79(10):1361 – 1369, 2006.

[167] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of AOSD '09*, pages 161–172, New York, NY, USA, 2009. ACM.

[168] Wei Zhang, Hua Yan, Haiyan Zhao, and Zhi Jin. A bdd-based approach to verifying clone-enabled feature models' constraints and customization. In *ICSR'08*, pages 186–199, Berlin, Heidelberg, 2008. Springer-Verlag.

[169] Christopher Zhong and Scott A. DeLoach. Runtime models for automatic reorganization of multi-robot systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 20–29, New York, NY, USA, 2011. ACM.